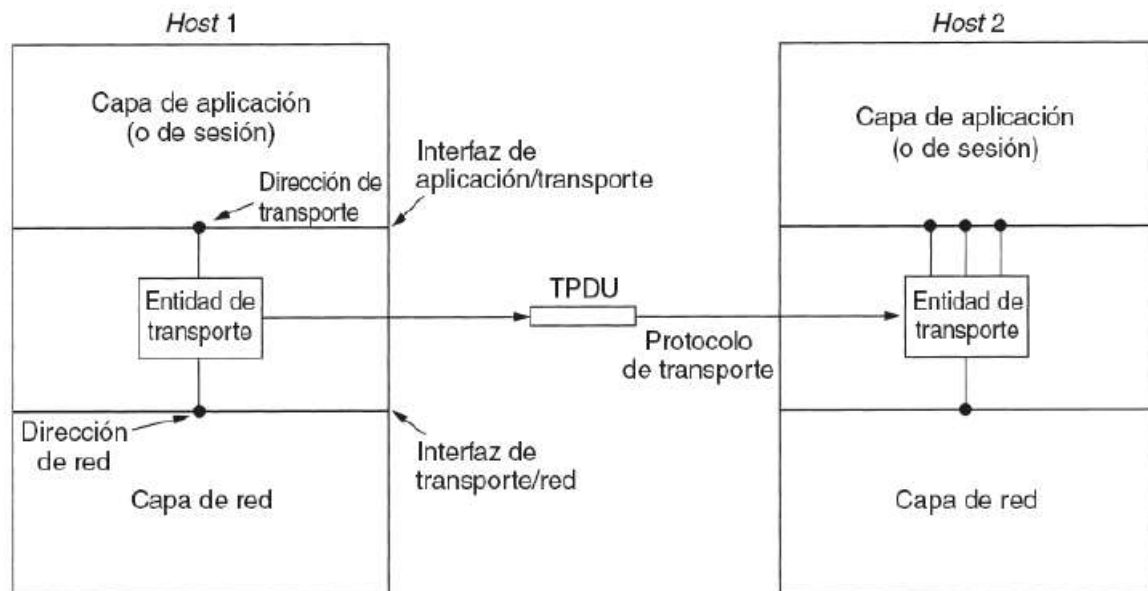


8. Nivel de Transporte

8.1. Introducción

La capa de transporte corresponde a protocolos de tipo end-to-end, es decir entre hosts que se desean comunicar, sin participación de los nodos intermedios (routers, switches, etc.).



TPDU = Transport Protocol Data Unit

Figura 45: El nivel de transporte en el modelo OSI.

Los protocolos end-to-end en subredes de datagramas se apoyan en la **capa de red**, que para IP es un servicio **best-effort**: puede descartar mensajes, desordenarlos, fragmentarlos y entregar duplicados. Además limita los mensajes en su tamaño, y los entrega en tiempos arbitrariamente largos.

Por lo tanto, los servicios end-to-end pueden ofrecer servicios para complementar esto:

- Garantía de entrega de mensajes (**confiabilidad**).
- Entrega de mensajes **en orden**.
- Entrega de a lo sumo **una copia** de cada mensaje.
- Soporte para mensajes de **largo arbitrario**.
- Soporte para **sincronización**.
- Posibilidad de que el receptor controle el flujo de datos del transmisor (**control de flujo**).
- Soporte para **múltiples procesos de nivel de aplicación** en el receptor.

8.2. Enlace de Datos versus Transporte

En un servicio de transporte, potencialmente:

- Se conectan **muchas máquinas diferentes**, por lo que requiere un establecimiento y finalización de conexión explícitos.
- Se tienen **diferentes RTT**, por lo que necesita mecanismos adaptativos para el time-out.
- Debe poder lidiar con **largos retardos** en la red.
- Se encuentra con **diferentes capacidades de recepción** en el destino, por lo que requiere contemplar nodos de diferentes capacidades.
- Trabaja con **diferentes capacidades de red**, requiriendo estar preparado para lidiar con **congestión**.

El nivel de enlace trabajaba con la comunicación entre nodos, mientras que el nivel de transporte es un mecanismo para **comunicación entre procesos de aplicación**. Esto implica lidiar con las limitaciones que tienen las capas inferiores (como el servicio best-effort de IP) y con los requerimientos que puedan tener las aplicaciones.

Al iniciar una comunicación entre procesos, estos deben especificar con qué proceso destino desean comunicarse. Para esto no sirven los PIDs que identifican los procesos en un sistema, ya que tienen alcance local, y son variables. Por lo tanto, en el nivel de transporte (al menos en TCP y UDP) se utilizan **puertos**: valores numéricos que identifican procesos. Esto, junto con la dirección IP del host, se utiliza para tener una caracterización unívoca del destino, y se suele implementar mediante **sockets**.

8.3. Transmission Control Protocol (TCP)

8.3.1. Conceptos Generales

- Ofrece un servicio **orientado a conexión**:
 - 3-way handshake para **setup**.
 - 2-2 o 4-way handshake para **liberación**.
- Provee un servicio de flujo de bytes (**stream-of-bytes**), en el sentido de que una aplicación escribe bytes, TCP envía segmentos y la aplicación del receptor lee bytes.
- Es un medio **full duplex** (dos flujos de bytes).
- Es **confiable**: establece una *conexión lógica entre sockets*, donde se utilizan ACKs, checksums y números de secuencia para manejar retransmisiones y ordenamiento.
- **Control de flujo**: evita que el transmisor *inunde* al receptor.
- **Control de congestión**: evita que el transmisor sobrecargue la red.

En la figura 46 se puede ver un esquema de una interacción vía TCP entre un cliente y un servidor: una aplicación del cliente escribe bytes, que se *bufferean* en el buffer de envío de TCP, y son agrupados en segmentos para enviarse al receptor. El TCP del receptor recibe los segmentos en su buffer de recepción, y le entrega los bytes a la capa superior de aplicación.

Notar la **diferencia entre control de flujo y control de congestión**: en el control de flujo, el objetivo es proteger al destinatario de un flujo de datos que no pueda soportar por su capacidad de recepción. En cambio, el objetivo del control de congestión es proteger a la red entera para que no se sobrecargue por las transmisiones.

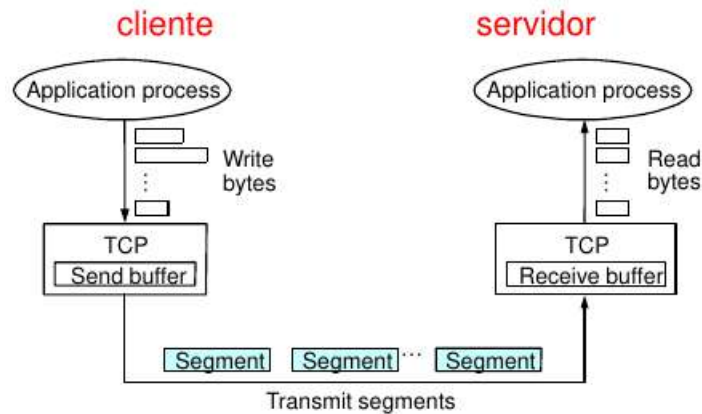


Figura 46: Comunicación cliente-servidor.

8.3.2. Maximum Segment Size

Es análogo al MTU (Maximum Transmission Unit) de la capa física, pero en este caso en la capa de transporte.

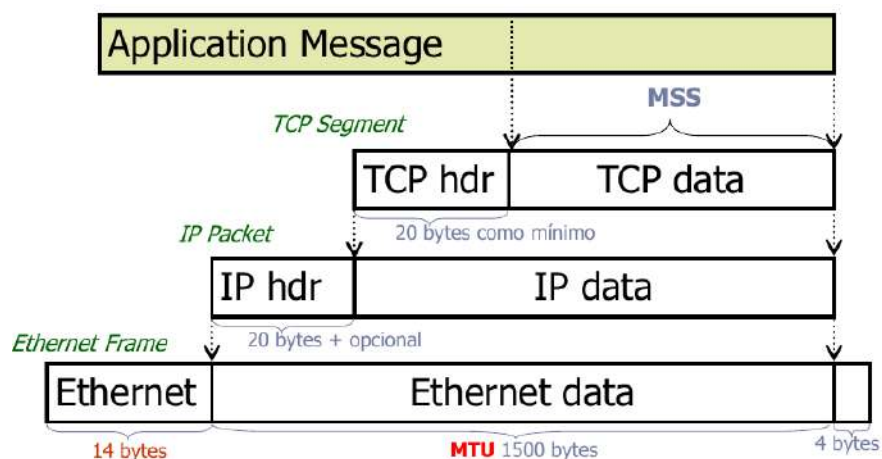


Figura 47: Las unidades de información manejadas por cada nivel tienen diferentes tamaños máximos (y diferentes nombres).

8.3.3. Segmento TCP

En la figura 48, se puede ver los campos presentes en un segmento del protocolo TCP:

- **SrcPort** y **DstPort** (16 bits ambos): puertos de origen y destino, respectivamente.
- **SequenceNum** (32 bits): número de secuencia, usado para identificar los segmentos a retransmitir o reordenar.
- **Acknowledgement** (32 bits): número usado para dar confiabilidad. Indica el siguiente byte que espera el receptor, implicando la confirmación de recepción para todos los bytes recibidos hasta ese momento.
- **AdvertisedWindow**: la ventana sugerida por el receptor, usada para control de flujo.

- **Checksum**: no sólo para el encabezado y los datos de TCP, sino también para parte del encabezado de IP (el *pseudo encabezado*: campos Protocol, SourceAddr, DestinationAddr), por motivos de eficiencia y utilidad.
- **HdrLen** (4 bits): indica la longitud del encabezado, en palabras de 32 bits.
- **UrgPtr** (Puntero a Urgente).
- **Options** (opciones): campo de longitud variable (por eso se necesita el HdrLen).
- **Flags**: distingue qué tipo de segmento es:
 - **URG** (urgent): es igual a 1 si el campo *Puntero a Urgente* está en uso.
 - **ACK** (acknowledge): es igual a 1 para indicar que el número de reconocimiento es válido (si es 0, el segmento no contiene un reconocimiento, por lo que ese campo será ignorado).
 - **PSH** (pushed data): indica al receptor que debe entregar los datos a la aplicación inmediatamente después de recibir el paquete (no esperar a que se llene el buffer).
 - **RST** (reset): usado para resetear una conexión que se volvió *confusa* por la caída de un host, o alguna otra razón.
 - **SYN** (sync): usado para establecer conexiones.
 - **FIN** (fin): usado para liberar conexiones.

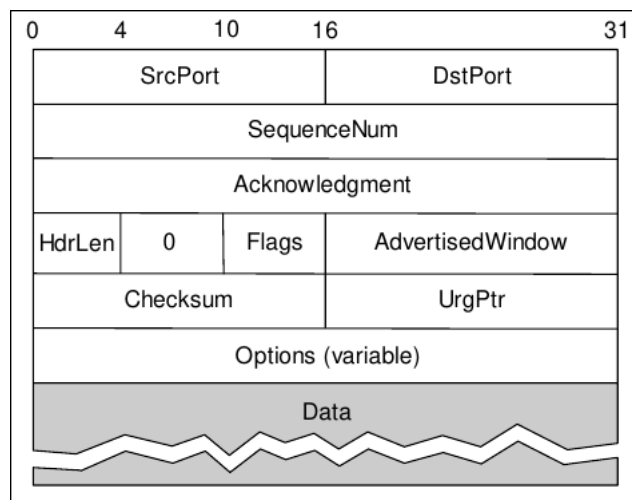


Figura 48: Formato de un segmento TCP.

Cada conexión TCP se identifica por la 4-upla: $\langle \text{SrcPort}, \text{SrcIPAddr}, \text{DstPort}, \text{DstIPAddr} \rangle$, es decir los números de puerto y dirección IP de origen y destino. Cada uno de estos dos pares es un **socket**.

Los puertos de origen se suelen elegir al azar, y la dirección de origen corresponderá a la que se le haya asignado a ese host.

Los puertos de destino suelen respetar ciertas recomendaciones acerca de dónde colocar cada servicio (e.g. páginas web en el puerto 80), y la dirección de destino será conocida por el transmisor (ya sea directamente o mediante una URL que será convertida en una IP).

8.3.4. Establecimiento de la Conexión

En la figura 49 se puede ver el esquema de un establecimiento de conexión. Cuando finaliza este 3-way handshake, ambos extremos de la comunicación saben que la conexión está activa, y el cliente podrá empezar a enviar datos.

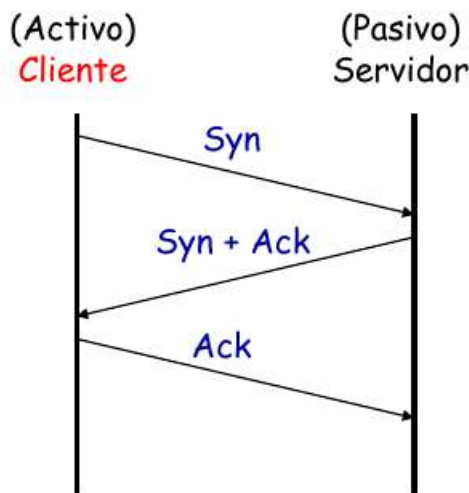


Figura 49: El procedimiento para establecer una conexión TCP se llama 3-way handshake.

Si bien TCP ofrece una conexión full duplex, usualmente es posible identificar un actor *pasivo* (servidor) y otro *activo* (cliente). El servidor debe hacer un *passive open* antes de poder recibir solicitudes. Esto consiste en ligarse a un puerto local, y empezar a *escuchar* conexiones allí. Luego, el cliente puede realizar un *active open*, que consiste en enviar un paquete TCP con el flag SYN encendido a dicho puerto.

El 3-way handshake tradicional consiste en los siguientes pasos:

- El servidor efectúa un LISTEN, y el cliente un CONNECT, acción que dispara un segmento SYN con un cierto número de secuencia inicial x .
- El servidor responde con SYN+ACK, indicando su número de secuencia y , y reconociendo el valor de x elegido por el cliente.
- Finalmente, el cliente reconoce el valor de y , quedando establecida la conexión.
- Si el servidor no dispusiera de un proceso escuchando en el puerto solicitado por el cliente, el sistema operativo responde con un segmento RST.

Existe otro escenario posible para el establecimiento de una conexión: la **apertura simultánea**. En este caso, ambos hosts responderán al SYN de su contraparte con un SYN+ACK, reconociendo el número de secuencia enviado. Al recibir los ACKs, cada host considerará que la conexión ha sido establecida.

8.3.5. Finalización de la Conexión

Existen diferentes maneras de efectuar la liberación de una conexión TCP, algunas más ideales que otras. La más clásica es la conocida como 4-way handshake, en la que el receptor (activo) decide que quiere terminar la comunicación, y el receptor envía los últimos datos que faltaran antes de acordar el cierre de la misma (ver figura 50).

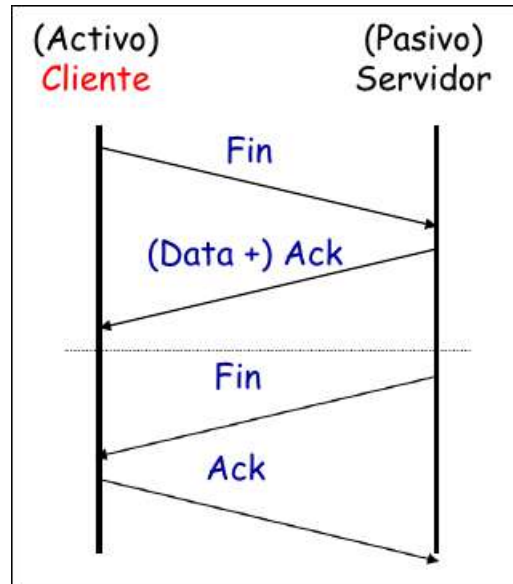


Figura 50: El procedimiento para el caso *feliz* para liberar una conexión TCP se llama 4-way handshake.

Al cerrar la conexión, TCP admite una **asimetría**: un extremo puede decidir cerrar su flujo de escritura, pero al mismo tiempo seguir recibiendo datos. Esto es lo que ocurre en el 4-way handshake:

- Un segmento **FIN** indica que el emisor no enviará más datos.
- El receptor debe reconocer ese **FIN** con un segmento **ACK**.
- Eventualmente, el receptor enviará su **FIN** cuando deje de recibir datos, y este debe ser reconocido por su contraparte.

Un escenario de cierre más común es el **simétrico**:

- El **FIN** inicial se responde directamente con **FIN+ACK**.
- El emisor del primer **FIN** responde con un **ACK**.

Por otro lado, al igual que con el inicio de conexión, el protocolo también especifica una situación de **cierre simultáneo**. En esta, cada interlocutor envía en forma independiente su **FIN** antes de que el **FIN** del otro sea correctamente recibido.

8.3.6. Ventana Deslizante

Es el mismo concepto de ventana deslizante (sliding window) aplicado para los protocolos punto a punto de la capa de enlace. Sin embargo, no se puede implementar el mismo mecanismo, ya que dependiendo de la intensidad de trabajo que esté realizando cada participante, su velocidad de procesamiento y recursos disponibles podrían variar.

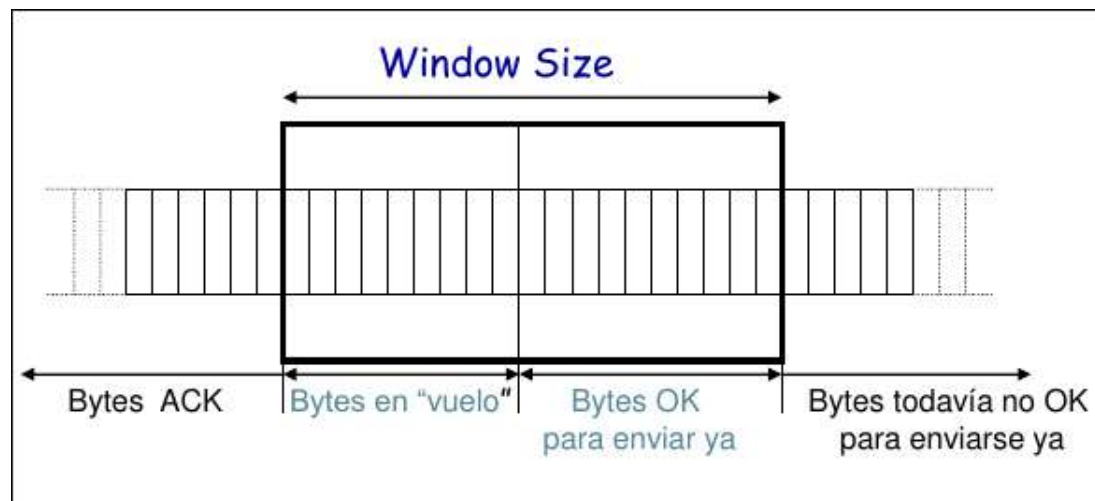


Figura 51: Esquema del sistema de sliding window para TCP.

La ventana tiene dos grupos de bytes: por un lado los bytes *en vuelo* (que ya fueron enviados, pero aún no se recibió el ACK), y los bytes listos para ser enviados.

Política de retransmisión: **GoBackN**.

Tamaño de la ventana: es *sugerido* (advertised) **por el receptor** (normalmente el servidor), y suele ser de entre 4 y 8 KB durante la fase de setup. El emisor no debe enviar más bytes que los sugeridos por el receptor.

Este protocolo garantiza **confiabilidad** (posibilidad de retransmitir), la aplicación recibe los datos **en orden** gracias a los números de secuencia y los números de ACK, y además fuerza el **control de flujo** entre receptor y transmisor.

Del **lado del emisor**, se tiene tres punteros para administrar los bytes enviados:

- **LastByteAked**: último byte reconocido por el receptor (de ahí hacia la *izquierda* todos los bytes fueron reconocidos).
- **LastByteSent**: último byte enviado.
- **LastByteWritten**: último byte escrito (de ahí a la *derecha* ningún byte fue aún generado).

Las relaciones a mantener son:

- $\text{LastByteAked} \leq \text{LastByteSent}$
- $\text{LastByteSent} \leq \text{LastByteWritten}$

Se *bufferean* los bytes entre **LastByteAked** y **LastByteWritten**.

Del **lado del receptor**, hay también tres punteros para administrar la recepción de información:

- **LastByteRead**: último byte leído.
- **NextByteExpected**: próximo byte esperado.
- **LastByteRcvd**: último byte recibido.

Las relaciones a mantener son:

- $\text{LastByteRead} < \text{NextByteExpected}$ (para mantener el orden).
- $\text{NextByteExpected} < \text{LastByteRcvd} + 1$ (un byte no puede ser leído por la aplicación a menos que éste se haya recibido, y también se hayan recibido todos sus precedentes).

Se *bufferean* los bytes entre NextByteExpected y LastByteRcvd . Este buffer podría crecer mucho, y es responsabilidad del receptor conocer su límite para hacérselo saber al emisor a través del campo `AdvertisedWindow` del encabezado de segmentos TCP.

8.3.7. Control de Flujo

- Tamaño del buffer de **envío**: `MaxSendBuffer`.
- Tamaño del buffer de **recepción**: `MaxRcvBuffer`
- Lado del **transmisor**:
 - $\text{LastByteSent} - \text{LastByteAcked} \leq \text{AdvertisedWindow}$.
 - $\text{EffectiveWindow} = \text{AdvertisedWindow} - (\text{LastByteSent} - \text{LastByteAcked})$.
 - $\text{LastByteWritten} - \text{LastByteAcked} \leq \text{MaxSendBuffer}$.
- Lado del **receptor**:
 - $\text{LastByteRcvd} - \text{LastByteRead} \leq \text{MaxRcvBuffer}$.
 - $\text{AdvertisedWindow} = \text{MaxRcvBuffer} - (\text{LastByteRcvd} - \text{NextByteRead})$.
- Se **bloquea la transmisión** si $(\text{LastByteWritten} - \text{LastByteAcked}) + y \leq \text{MaxSendBuffer}$, donde y es la cantidad de bytes que se desean escribir.
- Se envían ACKs desde el receptor en respuesta a la llegada de segmentos de datos.
- Cuando $\text{AdvertisedWindow} = 0$, el transmisor persiste enviando 1 byte (para evitar problemas de sincronización).

8.3.8. Máquina de Estados Finitos

En la figura 52, se puede ver una representación de la máquina de estados de TCP. Cada nodo representa un estado, y cada arco una transición entre estados. Estas transiciones se disparan por ciertos eventos, que pueden ser generados por el cliente o por el servidor, produciendo una reacción de la otra parte.

Estados:

- **CLOSED**: no hay conexión activa ni pendiente.
- **LISTEN**: el servidor espera una llamada.
- **SYN RCVD**: llegó una solicitud de conexión; se espera un ACK.
- **SYN SENT**: la aplicación comenzó a abrir una conexión.

- **ESTABLISHED**: estado normal de transferencia de datos.
- **FIN WAIT 1**: la aplicación dijo que ya terminó.
- **FIN WAIT 2**: el otro lado acordó liberar la conexión.
- **TIMED WAIT**: espera que todos los paquetes mueran.
- **CLOSING**: ambos lados intentaron cerrar la conexión simultáneamente.
- **CLOSE WAIT**: el otro lado inició una liberación de la conexión.
- **LAST ACK**: espera que todos los paquetes mueran.

El caso típico es que un cliente se conecta activamente a un servidor pasivo (indicado con líneas continuas para el cliente y punteadas para el servidor).

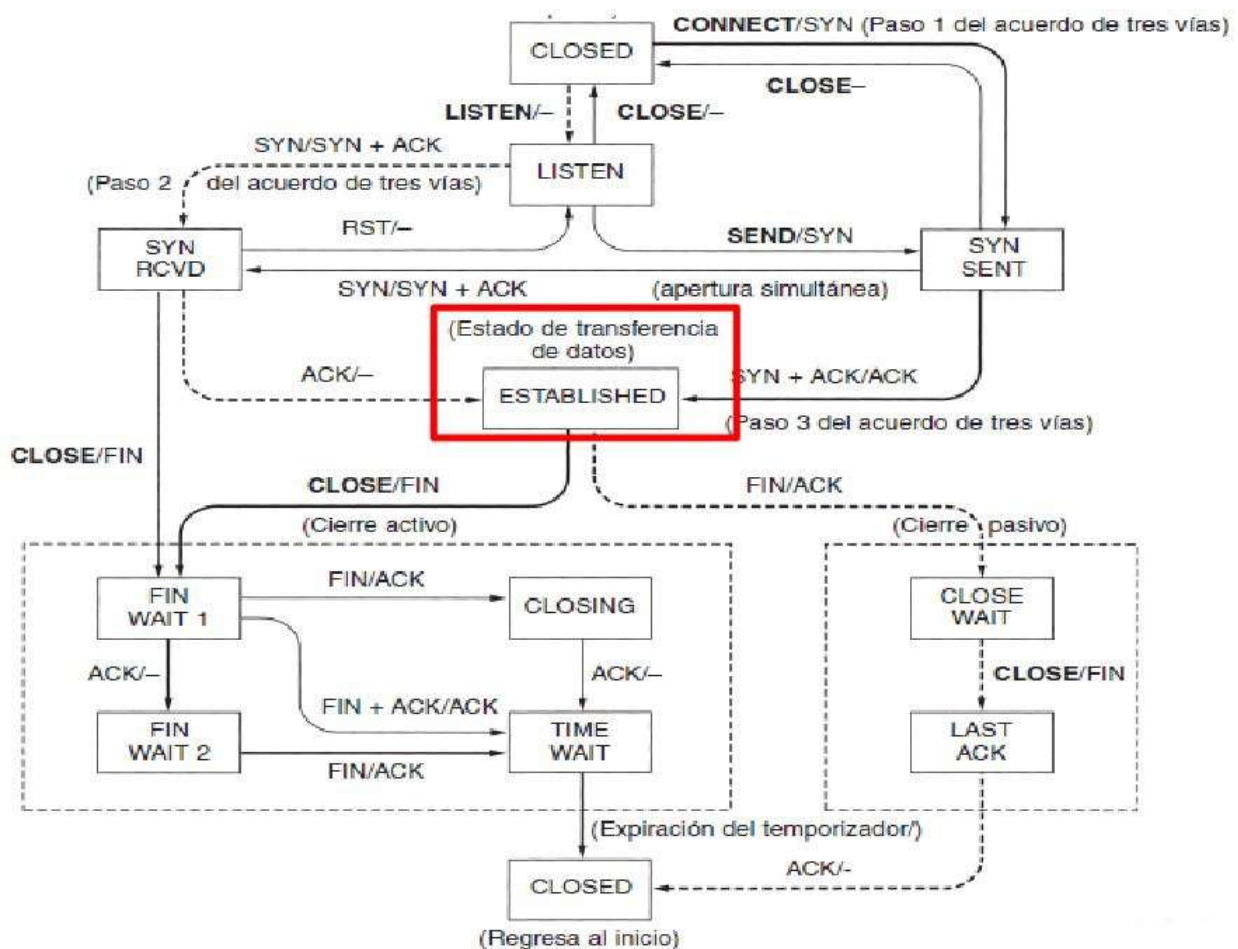


Figura 52: Máquina de estados de TCP. Los arcos continuos corresponden a acciones del cliente, y los punteados a acciones del servidor. Las etiquetas sobre los arcos siguen la forma Evento / Reacción. Si los eventos están en negrita, corresponden a una llamada de sistema iniciada por el usuario de la aplicación que corre en la capa superior; si no, corresponden a eventos generados por la llegada de un segmento TCP.

8.3.9. Retransmisión y Time-out

Retransmission Time-out (RTO): es el tiempo pasado el cual, si no llega un ACK del interlocutor, el emisor interpreta que hubo un problema (podría ser que los paquetes se perdieron

en el camino hacia el receptor, o que llegaran pero no volvieran los ACKs). En TCP, se asume que esa pérdida se dio por **congestión** en la red. Eso disparará una **retransmisión**.

Establecer el valor del RTO no es fácil, pues el **RTT puede variar** constantemente en la red (por congestión, por cambios en las rutas, etc.), y si el RTO no es lo suficientemente alto, no le dará tiempo a los paquetes/ACKs para llegar a destino.

Para esto se usan los algoritmos de **retransmisión adaptativa**. A grandes rasgos, el algoritmo original consiste en medir el RTT para cada par segmento+ACK, y calcular con ello el promedio ponderado del RTT. Una vez obtenido este valor, se fija el RTO en base a esa estimación de RTT (tomando el RTO como el doble del RTT estimado).

Esta idea puede presentar problemas al ponerla en práctica, como se puede ver en la figura 53. Los problemas más obvios tienen que ver con **intercumbre en los ACKs**.

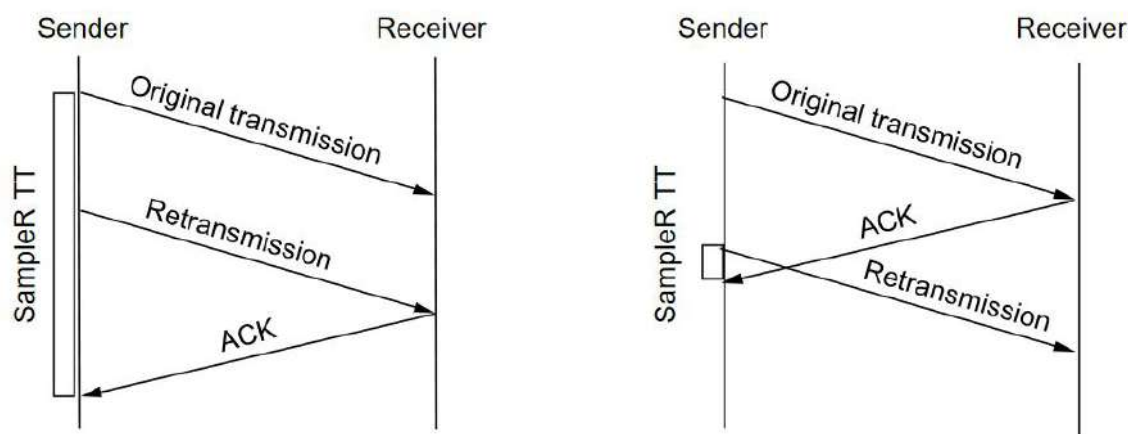


Figura 53: Dos escenarios problemáticos para el algoritmo de retransmisión adaptativa. A la izquierda, el RTT calculado para el par segmento+ACK no es preciso (es mayor que el real), pues hubo una retransmisión de por medio. A la derecha, el ACK para la transmisión original llega justo después de que se retransmita por un time-out, por lo que se calcula un RTT mucho menor al real.

Existen variantes del algoritmo de retransmisión adaptativa que atacan estos problemas: por ejemplo, el **algoritmo de Karn/Partridge** (1987), que propone no considerar el RTT cuando se retransmite, y duplicar el RTO luego de cada retransmisión (una idea similar al exponencial backoff visto anteriormente).

A su vez, esto evolucionó en otros algoritmos más sofisticados, como el **algoritmo de Jacobson/Karels** (1988), que propone una nueva forma de calcular el RTO considerando la varianza.

8.4. User Datagram Protocol (UDP)

8.4.1. Conceptos generales

UDP ofrece un mecanismo **liviano** y minimalista para la capa de transporte, de manera tal que sea posible entregar paquetes sin todo el overhead necesario (de tiempo, recursos, algoritmos, etc.) para sostener la infraestructura descrita para TCP.

A cambio, esto supone no poder contar con los beneficios de utilizar TCP (garantías en cuanto a confiabilidad, retransmisiones, ordenamiento, etc.).

8.4.2. Multiplexación mediante Puertos

Básicamente UDP implementa una multiplexación con puertos: los paquetes llegan a la capa de transporte, y UDP los separa para cada aplicación de la capa superior, que tiene un puerto asociado a ella (TCP también hace esto, sumado a sus otros mecanismos). Es decir, múltiples aplicaciones pueden usar la misma infraestructura de nivel de transporte, sin *molestarse* entre ellas.

En resumen, UDP **agrega demultiplexación** a IP.

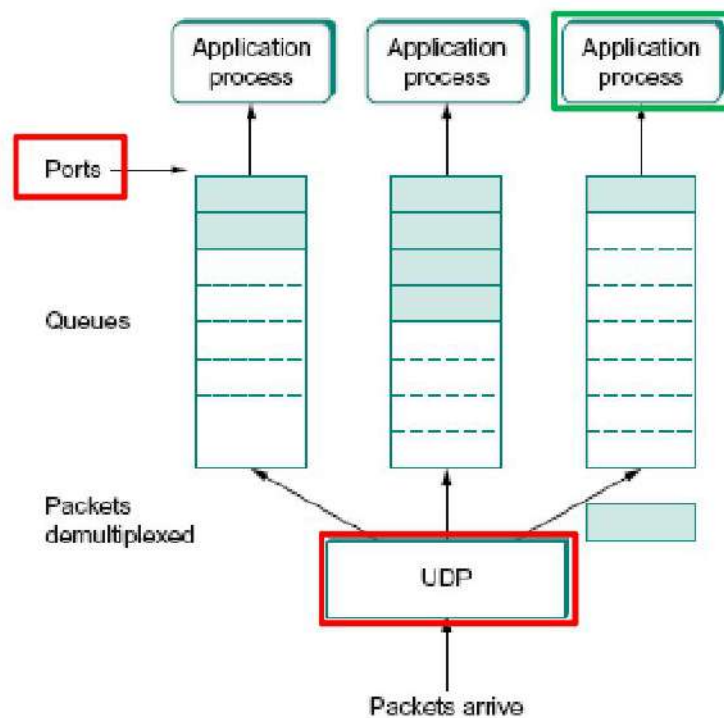


Figura 54: Esquema de la multiplexación de puertos para UDP, con una cola de buffering por cada proceso de nivel de aplicación.

8.4.3. Segmento UDP

En la figura 55 se puede ver que el contenido de un encabezado de UDP es muy escueto: contiene los puertos de origen y destino, la longitud en bytes del encabezado y los datos UDP, y un checksum para verificación de errores.

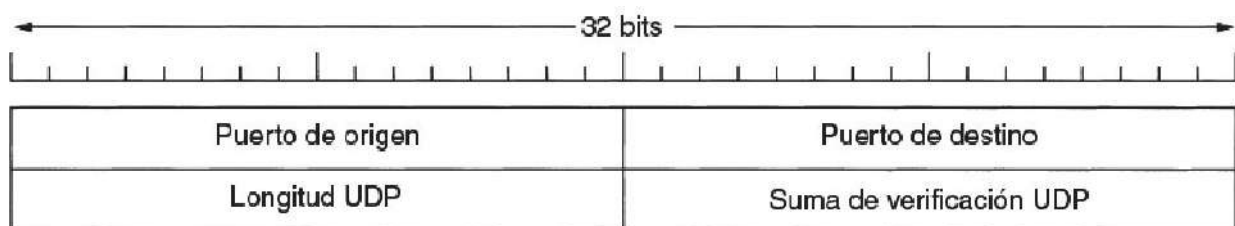


Figura 55: Formato de un segmento UDP.

8.4.4. Modelo de Servicio

UDP ofrece un modelo de comunicación **sin conexión**, con un mínimo de mecanismos: provee **checksums** para la integridad de los datos, y multiplexación a través de puertos.

Al no haber una conexión, no existen los handshakes utilizados en TCP para establecer y finalizar las comunicaciones. Esto expone la aplicación de usuario a una menor confiabilidad: **no hay garantías** de entrega, ordenamiento o unicidad para los datos.

Tampoco ofrece control de flujo ni control de congestión; ambas deben ser implementadas en otros niveles. No disponer de control de flujo puede producir que el destinatario de una comunicación se vea saturado y pierda información. Por otro lado, la falta de control de congestión tiene como resultado lo que se conoce como **tráfico egoísta**: se pueden congestionar las colas de entrada y salida de los routers.

UDP establece una comunicación **poco costosa** entre procesos, evitando la sobrecarga y las demoras propias de una entrega ordenada y confiable de mensajes. Además, soporta múltiples procesos de nivel de aplicación en cada máquina a través de la multiplexación mediante puertos.

Por lo tanto, se suele utilizar para situaciones en las que la verificación y corrección de errores no son necesarias, o se realizan en el nivel de aplicación (por ejemplo, aplicaciones que requieren que todo llegue lo más rápido posible, donde es preferible perder paquetes antes que enfrentar demoras por las retransmisiones). Por ejemplo, se puede usar en conjunto con RTP (Real-time Transport Protocol), un protocolo utilizado para audio y video sobre redes IP; también se utiliza para sistemas de tiempo real, como puede ser una co-simulación (dos computadoras corriendo una simulación distribuida en tiempo real, simultáneamente).

8.5. Congestión

La congestión es descubierta e investigada a finales de los 80 en la red precursora de Internet. Se detectó un deterioro importante del throughput (congestion collapses), que no podía ser explicado por fallas en los nodos que formaban parte de la red.

8.5.1. Administración de Buffers

En un contexto en donde se aplica la **multiplexación estadística** para organizar la asignación de ancho de banda a los participantes de una red, la llegada de los paquetes no es un evento predecible, sino que es aleatorio.

Por lo tanto, deja abierta la posibilidad de que los buffers se llenen, lo cual tiende a suceder por la presencia de *cuellos de botella*. Estos **buffers** se utilizan para reducir las pérdidas de paquetes, pero también incrementan el **Delay** total en la comunicación a través de la red (relacionado con el tiempo de encolamiento).

Si un buffer se **sobrecarga** (overflow), se descartan paquetes, lo cual obligará a realizar **retransmisiones** (al menos en el caso de TCP), degradando la calidad de servicio (QoS). Para garantizar esta calidad, se debe **reservar espacio** en los buffers. Sin embargo, esto se opone al objetivo de una administración descentralizada y *automática* de los recursos.

Si los buffers del receptor de una transmisión se llenan, es necesario que el emisor se pueda enterar y sea capaz de reaccionar para no agravar el problema. Y la idea es hacer esto de manera distribuida, sin un control central.

8.5.2. Definición y Soluciones

La **congestión** es el **estado de sobrecarga *sostenida*** de una red, donde la demanda de recursos (enlaces, buffers) está al límite o por encima de su capacidad.

Las consecuencias son perceptibles en términos de una **degradación de la calidad de servicio**.

Las soluciones posibles para este fenómeno vienen por distintos lados, por ejemplo:

- Sobredimensionamiento (overprovisioning).
- Diseño cuidadoso.
- Control proactivo (evitar la congestión, control preventivo): decrementar la carga.
 - ¿Cómo saber qué usuarios están sobrecargando la red?
 - Asignar un ancho de banda limitado a cada organización.
 - Premiar el *uso parejo*, y castigar el *uso excesivo*.

El problema en común para todas estas soluciones es que **no escalan**.

Capa	Políticas
Transporte	<ul style="list-style-type: none"> • Política de retransmisión • Política de almacenamiento en caché de paquetes fuera de orden • Política de confirmaciones de recepción • Política de control de flujo • Determinación de terminaciones de temporizador
Red	<ul style="list-style-type: none"> • Circuitos virtuales vs. datagramas en la subred • Política de encolamiento y servicio de paquetes • Política de descarte de paquetes • Algoritmo de enrutamiento • Administración de tiempo de vida del paquete
Enlace de datos	<ul style="list-style-type: none"> • Política de retransmisiones • Política de almacenamiento en caché de paquetes fuera de orden • Política de confirmación de recepción • Política de control de flujo

Figura 56: Políticas que influyen en la congestión, en distintas capas.

La tabla de la figura 56 sugiere que el problema de la congestión es difícil de solucionar: no es algo que se arregle apuntando a un aspecto o un nivel específicos de las capas del modelo OSI.

8.5.3. Análisis de Congestión

Para realizar un análisis formal de la congestión, se usarán conceptos de **teoría de colas**.

Sistema M/M/1: es un sistema cola-servidor en el cual el número de llegadas de paquetes es markoviano (proceso de Poisson), y el tiempo de servicio de paquetes también es markoviano (proceso exponencial). Existe un único servidor, y la cola tiene capacidad infinita:

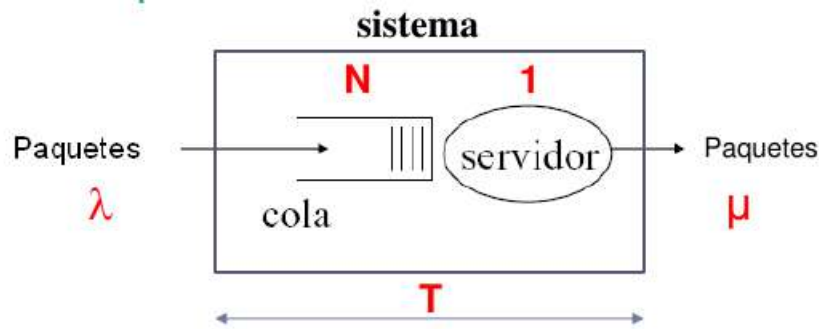


Figura 57: Esquema de un sistema M/M/1. N es la cantidad de paquetes que entran en el buffer, T es el tiempo medio total que está un paquete en el sistema (tanto esperando como siendo procesado). λ es la **tasa de entrada** de paquetes a procesar en el sistema (por segundo), y μ es la **tasa de servicio** de paquetes procesados por el sistema (por segundo).

En base a las métricas mencionadas, se tiene que $1/\lambda$ es el tiempo medio entre llegadas, y $1/\mu$ el tiempo medio de servicio. A su vez, se define $\rho = \lambda/\mu$ como la **intensidad del sistema**.

El sistema puede alcanzar un **estado estacionario** si y sólo si $\rho < 1$. Si la tasa de entrada es mayor que la tasa de servicio (i.e. $\rho > 1$), se desborda el sistema: hay **congestión**.

Usando esta representación markoviana, se pueden obtener resultados vía un análisis probabilístico:

$$\begin{aligned} E(N) &= \frac{\rho}{1 - \rho} \\ &= \lambda E(T) \end{aligned}$$

Entonces:

$$\begin{aligned} E(T) &= \frac{E(N)}{\lambda} \\ &= \frac{\rho}{(1 - \rho) \lambda} \\ &= \frac{1}{\mu - \lambda} \end{aligned}$$

Esto es importante porque lo que se quiere es que el buffer no se sature; por lo tanto es útil conocer la esperanza de N (cantidad de paquetes en el buffer), y de T (tiempo que está un paquete en el sistema).

Este modelo se puede ampliar a más servidores: sistema M/M/m, siendo m la cantidad de servidores. Pero no siempre agregar m servidores reducirá el problema de la congestión proporcionalmente (el tiempo de respuesta en función de la intensidad del sistema siempre tiene un crecimiento exponencial, lo que se puede mejorar es que tan rápido se llega al punto en el que el tiempo es demasiado alto).

8.5.4. Métricas de Detección

Existen varias métricas posibles para **detectar congestión**:

- Porcentaje de paquetes descartados por falta de espacio en buffer.

- Longitud media de una cola (buffer).
- Cantidad de paquetes que generan time-out y son retransmitidos.
- Demora promedio de los paquetes (average packet delay).
- Desvío estándar de la demora de los paquetes (standard deviation of packet delay).

8.5.5. Causas de la Congestión

Existen muchos factores que pueden producir congestión en una red. Algunos de ellos son:

- Inundaciones con tráfico destinado a una misma línea de salida.
- Procesadores lentos.
- Problemas con software de ruteo.
- Cuellos de botella en distintas partes del sistema *aguas abajo*.

Además, el efecto de la congestión tiende a **realimentarse** y **empeorar** con el tiempo. En el límite se puede alcanzar lo que se conoce como un **colapso de congestión**, donde el goodput tiende a cero.

8.5.6. Control de Congestión

Informalmente, se puede pensar la congestión como *demasiadas fuentes utilizando una red compartida, enviando demasiados datos demasiado rápido como para poder ofrecer una buena calidad de servicio*.

Los **síntomas** típicos pueden ser la **pérdida de paquetes** (debido a la saturación de buffers en los routers o switches) y los **retardos crecientes** (por el encolamiento en dichos buffers).

En la materia se asumirá que los routers son de tipo drop tail (i.e. se descartan paquetes cuando se llena la capacidad) y FIFO (i.e. los paquetes se transmiten en el orden en que llegan). Combinados, pueden producir una sincronización global cuando los paquetes descartados provienen de distintas conexiones no sincronizadas entre sí.

El **control de la congestión en la red** ataca un problema muy distinto al **control de flujo en el receptor**:

- Control de congestión: debería poder evaluar la capacidad de una **subred completa** para transportar un determinado tráfico *agregado*. Es decir, la congestión tiene un **efecto global** en la red, que involucra a todos los hosts y routers. La idea es **evitar que los transmisores sobrecarguen el interior de la red**.
- Control de flujo: controla el **tráfico punto-a-punto** entre un transmisor y un receptor particulares. La idea es **prevenir que los transmisores sobrecarguen a receptores lentos**.

Control de congestión: es el esfuerzo hecho por los nodos de la red para **prevenir o responder a sobrecargas de la red**, que conducen a pérdidas no controladas de paquetes.

Las dos maneras más obvias de encarar esto son:

- **Pre-asignar recursos** (ancho de banda y espacio de buffers) para evitar la congestión. Esto conduce a la **subutilización** del medio.
- **Liberar recursos** y controlar la congestión sólo si ocurre. La pregunta es **¿a quién perjudicar?**

Una solución es permitir el uso de recursos de la red en forma **equitativa**. Cuando ocurran problemas, sus efectos se comparten entre todos los usuarios, en vez de causar enormes dificultades para algunos.

El control de congestión se implementa en dos puntos:

- **Hosts en los extremos** de la red (protocolo de transporte).
- **Routers dentro** de la red (disciplina de encolamiento).

8.5.7. Criterios de Evaluación

El objetivo es que la red sea utilizada **eficientemente** y en forma **equitativa**. Un buen indicador para la eficiencia es la **Potencia**: throughput [b/s] / delay [s]:

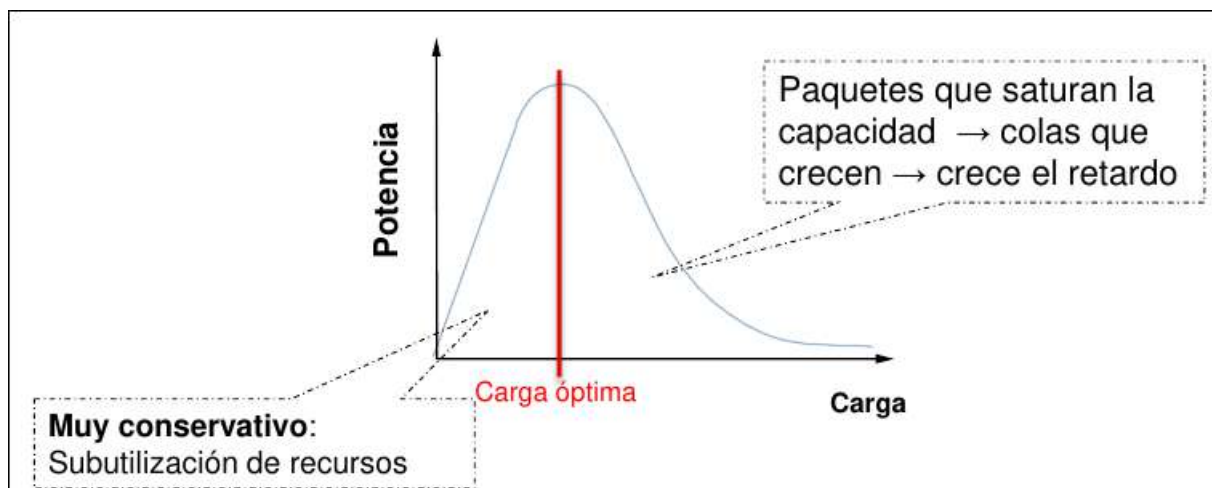


Figura 58: Un gráfico utilizado para analizar la potencia en función de la carga en la red (ρ).

La noción de potencia intenta combinar dos aspectos: el throughput y el tiempo de respuesta en la red; ambos en función de la carga.

- El throughput crece linealmente con la carga (se llenan los buffers) pero llega a una *rodilla* seguida de un *acantilado*, es decir: a partir de un momento decrece súbitamente cuando la carga alcanza la capacidad de la red. Una vez que los buffers empiezan a sobrecargarse, ocurre la pérdida de paquetes. Incrementar la carga más allá de este punto aumenta más la probabilidad de perder paquetes.
- El tiempo de respuesta crece de manera exponencial cuando aumenta la carga (como ya se vio en gráficos anteriores). Bajo cargas extremas, el tiempo de respuesta tiende a infinito (y el throughput a cero): este es el punto de colapso de congestión.

En cuanto a la **equidad** (que los recursos sean compartidos equitativamente), existen distintos criterios para evaluarla. Por ejemplo, el indicador de equidad de Jain.

8.5.8. Congestión y Calidad de Servicio

Si las redes nunca sufriesen de congestión, sería fácil brindar una buena Calidad de Servicio (QoS): simplemente sobredimensionar los enlaces.

En la realidad la congestión ocurre, y para proveer QoS en estos escenarios es necesario tener mecanismos que permitan dar un trato diferenciado a cierto tráfico preferencial, y cumplir los SLA (Service Level Agreement, acuerdos de servicio que suelen ser definidos en el momento de negociación de contrato con el proveedor de servicio de Internet).

8.5.9. Teoría de Control

Los mecanismos de control de congestión que funcionan en Internet son mecanismos de **control automático** (la manera en la que funciona la mayoría de sistemas autocontrolados).

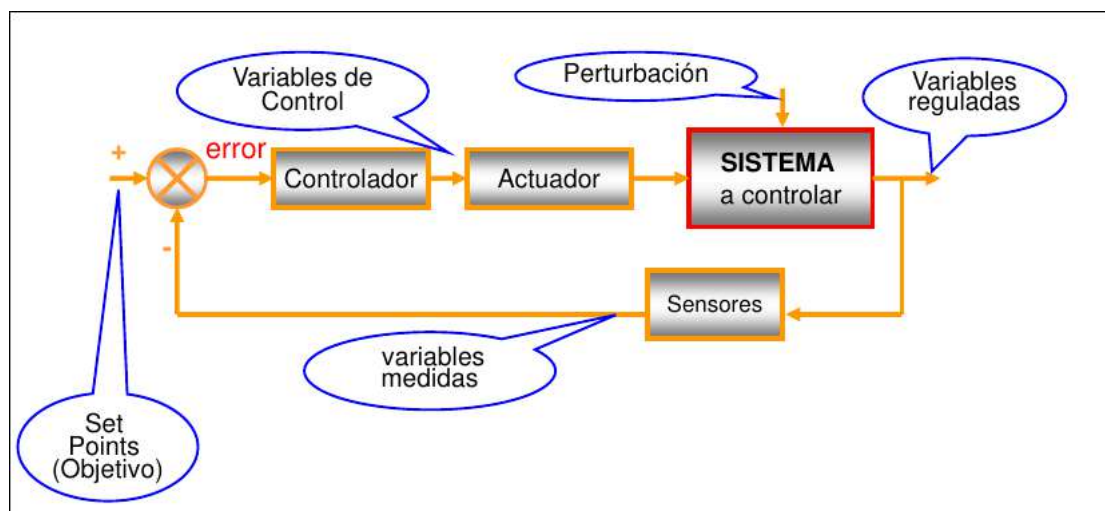


Figura 59: Esquema típico de control automático de lazo cerrado; usado en disciplinas variadas como electrónica, hidráulica, ingenierías química, eléctrica, etc.

El sistema de control de congestión de TCP implementa un esquema de control automático de **lazo cerrado** (ver figura 59) con **realimentación implícita**, donde intervienen:

- El sistema a controlar (los buffers).
- Una perturbación que saca al sistema del equilibrio que se desea obtener.
- Un objetivo (set point) a conseguir (es el llenado promedio de un buffer).
- Variables reguladas, cuyo estado se medirá con sensores; esos resultados de las mediciones se comparan con el objetivo, obteniendo la diferencia entre ellos (el error).
- Un algoritmo controlador, que manipulará variables de control acorde al error.
- Un actuador, que toma las variables de control mencionadas y actúa sobre el sistema (descarta paquetes).

Según la taxonomía de Yang y Reddy (ver figura 60), los algoritmos de control de congestión se pueden clasificar en **lazo abierto** y **lazo cerrado**. A su vez, estos últimos se pueden clasificar de acuerdo a cómo hacen la **retroalimentación**.

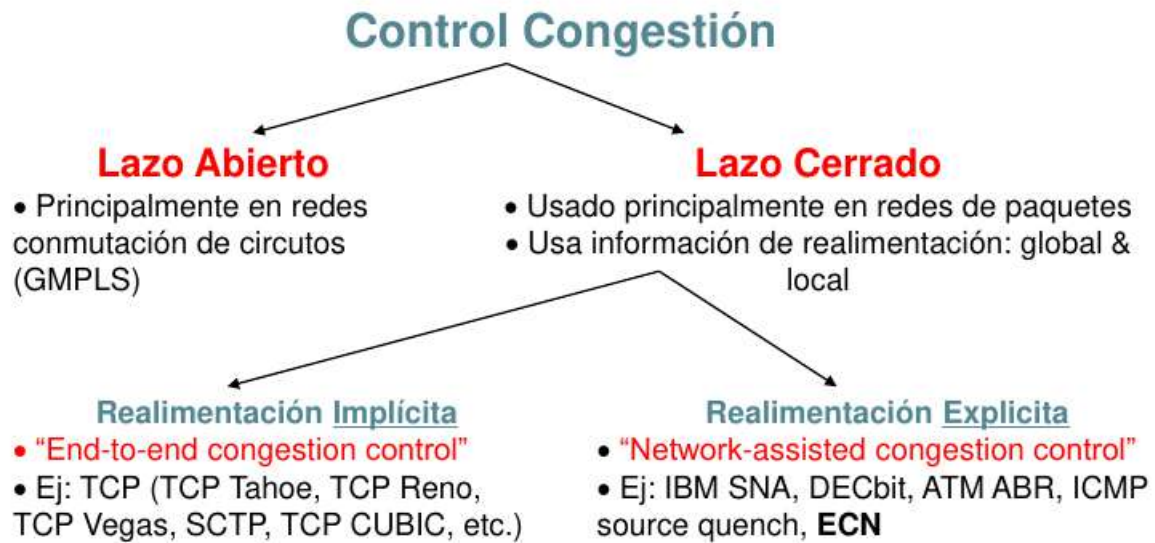


Figura 60: Taxonomía de Yang y Reddy (1995).

Control de Congestión con Retroalimentación implícita y explícita:

- **Implícita:** la red **descarta paquetes cuando se aproxima la congestión**; el emisor **infiere la congestión** en forma implícita (usando time-outs, ACKs duplicados, etc.). Por ejemplo, TCP End-to-End Congestion Control.
- **Explícita:** los componentes de red (switches, routers, etc.) proveen una **indicación explícita de la congestión** a los emisores (*packet marking*); provee información más precisa a las fuentes, pero es más complicado de implementar, y todos los routers de la ruta de un paquete deben tener esta opción activada, caso contrario no funciona (requiere cooperación explícita entre fuentes y componentes de red). Por ejemplo, DECbit, ECN, etc.

8.5.10. Random Early Detection (RED)

Esta es la implementación concreta de la estrategia de control de congestión de lazo cerrado con realimentación implícita, que trabaja en colaboración con protocolos como TCP (**no es parte de TCP**, opera en los routers, pero asume que en los extremos hay un protocolo TCP, que será reactivo a lo que haga RED en los routers: **comunicación implícita**).

Fundamentos y características:

- El objetivo del algoritmo RED es **evitar la congestión** (congestion avoidance), manteniendo el tamaño medio de las colas (buffers de los routers) en niveles *relativamente bajos*.
- No necesita que los routers mantengan información acerca del estado de las conexiones (comunicación **implícita**).
- Fue diseñado para trabajar en **colaboración** con mecanismos de control de congestión de la **capa de transporte** (e.g. TCP).

- Es una técnica de las conocidas como AQM (Active Queue Managment, *administración activa de colas*).

Estrategia:

- **Notificación implícita de la inminencia de la congestión**, a través de descartar el paquete (luego en TCP ocurrirán los timeouts).
- **Descarte aleatorio temprano**: no espera a que se llene el buffer, sino que descarta cada paquete según **alguna probabilidad de descarte, cada vez que la cola excede algún nivel de llenado**.

Algoritmo:

- Calcula el **largo promedio de la cola**. Esto corresponde a la etapa de **producir la medición** de las variables reguladas en el esquema de control automático. El cálculo del largo de la cola se hace teniendo en cuenta dos componentes con pesos (**Weight**) asociados:
 - **SampleLen**: tamaño instantáneo de la cola, actualizado cada vez que llega o sale un paquete.
 - **AvgLen**: versión *suavizada*, promediada de **SampleLen** (tiene cierto componente histórico).
- Compara ese largo con dos **umbrales** del tamaño promedio de la cola (**MinThreshold** y **MaxThreshold**). Esto corresponde a **comparar las mediciones con los objetivos** en el esquema de control automático. A su vez, la parte de **actuar sobre el sistema** se realiza a través del descarte de paquetes, gobernado por las siguientes comparaciones:
 - Si $\text{MinThreshold} \geq \text{AvgLen}$, entonces se **encola** el paquete.
 - Si $\text{MinThreshold} < \text{AvgLen} < \text{MaxThreshold}$, entonces se calcula una probabilidad p , y se **descarta** el paquete entrante con esa **probabilidad p** .
 - Si $\text{MaxThreshold} \leq \text{AvgLen}$, entonces se **descarta** el paquete entrante.

Observaciones:

1. RED **descartará paquetes incluso cuando estos tengan espacio** para encolarse en los buffers. Establecer el cálculo del largo promedio, y los umbrales máximo y mínimo no es trivial, y corresponde a una decisión de diseño que impactará en qué tan bien funciona el mecanismo.
2. Las comparaciones con los umbrales se hacen sobre el **promedio** del tamaño del buffer. Esto significa que el llenado real de esta cola en un momento dado puede ser completo (y mayor al umbral máximo), pero lo que no puede ocurrir es que el promedio de llenado sea superior a ese umbral. Esto se hace de esta manera para poder **absorber ráfagas** de paquetes. La probabilidad de descartar un paquete de un flujo particular de paquetes es aproximadamente proporcional a la porción del ancho de banda que dicho flujo está obteniendo.
3. La probabilidad p suele tener un valor máximo de 0.02: en este caso el router descartará aproximadamente uno de cada cincuenta paquetes.

4. Si el tráfico tiene muchas **ráfagas**, el valor de **MinThreshold** debería ser lo suficientemente grande como para permitir que la utilización del enlace sea mantenida en un nivel aceptable.
5. La **diferencia entre los dos umbrales** debería ser más grande que el incremento típico del largo de cola promedio calculado en un RTT (a veces se toma $\text{MaxThreshold} = 2 \times \text{MinThreshold}$).

8.5.11. Flow Random Early Detection (FRED)

RED tiene problemas de **imparcialidad** con las conexiones de **baja velocidad**: cuando se alcanza el umbral, RED descarta paquetes aleatoriamente sin considerar si se trata de una conexión que está usando más de su cuota de recursos o menos.

Es por esto que se implementó una mejora que se convirtió en **FRED** (*Flow Random Early Detection*) en el que se controla a los usuarios “bandidos”: mantiene umbrales y tasas de ocupación del buffer para cada flujo activo.

La desventaja notoria de esto es que necesita guardar información por cada flujo, produciendo un alto costo en los routers (debe mirar la fuente, destino, puertos y protocolo del paquete por cada paquete que llega).

8.5.12. Control de Congestión en TCP

Definiciones:

- Sender Maximum Segment Size (SMSS): máximo tamaño que pueden tener los segmentos enviados.
- Receiver Maximum Segment Size (RMSS): máximo tamaño de segmento que puede recibir el receptor.
- Full-Sized Segment: segmento con SMSS bytes.
- Receiver Window (RWND): última Advertised Window recibida.
- Congestion Window (CWND): variable que limita la cantidad de datos que puede enviar el emisor. Es una estimación de la cantidad de información que se puede introducir en la red sin que se degrade su performance.
- Initial Window (IW): valor de CWND después del handshake de establecimiento de la conexión TCP.
- Loss Window (LW): valor de CWND después de un time-out.
- Restart Window (RW): valor de CWND después de un período ocioso (idle).
- MaxWindow = $\min(\text{RWND}, \text{CWND})$: limitada por el host o la red.
- EffectiveWindow = $\text{MaxWindow} - (\text{LastByteSent} - \text{LastByteAcked})$: cuántos bytes se puede despachar.
- Flight Size = $\text{LastByteSent} - \text{LastByteAcked}$.

- Slow Start Threshold (SSTHRESH): umbral que define si se usa slow start o congestion avoidance. Si es mayor que CWND, se usa slow start, y si es menor, congestion avoidance.

ACK Clock: la emisión de nuevos segmentos se ve regulada por la tasa de llegada de los ACKs (el protocolo es self-clocking). Esto ayuda a tener bajos niveles de pérdida y delay en la red.

El mecanismo de control de congestión implementado por TCP se suma al sistema de sliding window usado para control de flujo:

- Se define una nueva variable Congestion Window (CWND), que representa cuántos datos la red está disponible a aceptar antes de congestionarse (es análogo a la Advertised Window, pero en lugar de decir cuántos datos puede recibir el receptor, dice cuántos se aguantaría la red).
- Ahora el emisor tiene que enviar datos teniendo en cuenta que no sólo no debe pasarse de la Advertised Window, sino también de la Congestion Window.
- $\text{MaxWindow} = \min(\text{CongestionWindow}, \text{AdvertisedWindow})$, y $\text{EffectiveWindow} = \text{MaxWindow} - \text{FlightSize}$.
- Para determinar el tamaño de la ventana de congestión, TCP *va tanteando cuánto puede mandar sin que haya congestión*.
- TCP asume que hay congestión cuando ocurre un time-out de un paquete (es decir, un byte no es reconocido luego de un RTO), ya que esto es posiblemente porque el paquete fue descartado en algún router intermedio).
- Comienza con una Congestion Window de tamaño MSS, y va incrementándola hasta que eventualmente ocurre un time-out.
- Incrementa CWND en cada ACK de nuevos datos. Cada ACK nuevo es tomado como un signo de mayor capacidad disponible de la red.

ACK duplicados: un ACK es duplicado si:

- El receptor del ACK tiene datos sin confirmar.
- El ACK no tiene datos.
- No están encendidos los flags SYN ni FIN.
- El número de ACK es igual al máximo ACK recibido.
- La Advertised Window es igual a la última recibida.

Cuando RED descarta un paquete, se genera una **señal indirecta para TCP**, ya que el próximo paquete que no sea descartado (retransmisión) producirá un ACK duplicado en el otro extremo. Consecuentemente, el TCP del emisor **adaptará su ventana** para reaccionar a la nueva situación del tráfico, de la cual se enteró indirectamente.

Esto se organiza de acuerdo a la siguiente idea básica. Por cada RTT:

- Si se **recibe** un ACK, aumentar el tamaño W de la ventana, que pasa a ser $W + 1/W$ (**additive increase**).

- Si se **pierde** un ACK, reducir el tamaño W de la ventana, que pasa a ser $W/2$ (**multiplicative decrease**).

La idea del proceso de Additive Increase / Multiplicative Decrease (AIMD) es **salir lo antes posible de un estado de congestión**.

Todos los nodos de la red, sin estar comunicados entre sí, se comportarán de esta manera; cuando RED da la señal implícita de que hay congestión, recortan su ventana (con lo cual los buffers tienen tiempo para desagotarse), y cuando se reciben bien los paquetes, la van abriendo de a poco.

Algoritmos de control de Congestión: estos conviven en TCP y se aplican dependiendo de la situación (ver figura 61):

- **Slow Start**: comenzar enviando pocos datos.
 - Inicialmente, $CWND = IW = 2 \times SMSS$, y $SSTHRESH = MaxAdvertisedWindow$.
 - Si $CWND < SSTHRESH$, entonces a la $CWND$ se le suma $\min(N, SMSS)$ por cada ACK, donde N es la cantidad de bytes reconocidos por el ACK.
 - Si se pierde un paquete, $SSTHRESH$ pasa a ser $\max(FlightSize/2, 2 \times SMSS)$.
 - Se usa la llegada de ACKs como retroalimentación positiva.
 - Es un crecimiento exponencial, mucho más rápido que el Additive Increase, pero no tan brusco como tener una ventana de tamaño fijo.
 - La idea es que AIMD es útil cuando se está cerca de llegar al punto de congestión. Por el contrario, si se empieza con una ventana de congestión chica y se la aumenta lentamente, probablemente se tarde mucho en llegar al punto de congestión, desperdiciando capacidad de la red.
 - Se usa **al principio de una conexión**, para acelerar la convergencia al punto de congestión y pasar a AIMD; y además se usa **cuando ocurre un time-out**, hasta que la ventana de congestión tenga tamaño $SSTHRESH$.
- **Congestion Avoidance**: aumentar un $SMSS$ por RTT.
 - Si $CWND > SSTHRESH$, entonces a la $CWND$ se le suma $SMSS \times SMSS/CWND$ por cada ACK.
 - Esto es el Additive Increase.
 - Ante un time-out, se fijan $CWND = LW$, y $SSTHRESH = \max(FlightSize/2, 2 \times SMSS)$, y se comienza de nuevo con Slow Start.
 - Se usa el time-out como retroalimentación negativa.
- **Fast Retransmit**: no esperar al time-out para recuperarse de un error.
 - La idea es detectar una pérdida antes de un time-out, para no tener que volver a Slow Start después de cada time-out.
 - El receptor emite un ACK por cada segmento recibido, aún si es un segmento fuera de orden. Es decir, el receptor emite ACKs duplicados (ACKs de un mismo número de secuencia).
 - Cuando el emisor recibe 3 ACKs duplicados de un mismo byte, dispara la retransmisión y asume que hubo congestión, poniendo $SSTHRESH = CWND / 2$ y $CWND = MSS$, y hace Slow Start.

- El algoritmo es:
 - Al tercer ACK duplicado, $SSTHRESH = \max(\text{FlightSize}/2, 2 \times SMSS)$, y $CWND = SSTHRESH + 3 \times SMSS$.
 - Mientras no se reconozcan nuevos datos, sumarle un SMSS al CWND por cada ACK duplicado.
 - Se termina cuando llega el primer ACK que reconoce nuevos datos, estableciendo $CWND = SSTHRESH$.
 - Luego se continúa con Congestion Avoidance.
 - Se usa el tercer ACK duplicado como retroalimentación negativa.
- **Fast Recovery:** cuando se hace Fast Retransmit, en lugar de poner $CWND = MSS$ y hacer Slow Start hasta $SSTHRESH$, poner directamente $CWND /= 2$ y hacer AIMD. Sólo hacer Slow Start si efectivamente ocurre un time-out.

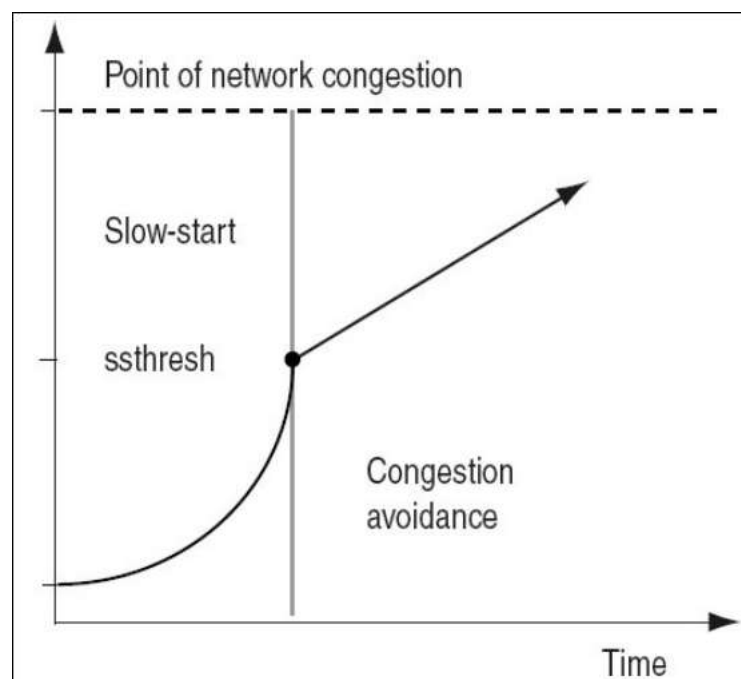


Figura 61: Slow Start + Congestion Avoidance.

8.5.13. Performance de TCP

Para hacer un análisis de la performance de TCP con control de congestión (Mathis et al., 1997), la idea fue hallar una expresión para el steady state throughput (el throughput *promedio*) en función del RTT y la probabilidad p de pérdida de paquetes.

Utilizando esto, con ciertas simplificaciones y usando el área bajo las curvas generadas por graficar el tamaño de la ventana de TCP en función del RTT, se calcula la cantidad de paquetes transmitidos por ciclo, y la duración de un ciclo.

Además, se llega a una relación directa entre el tamaño de la ventana y la probabilidad de pérdida de paquetes; con lo cual se puede llegar a una expresión para el throughput y el ancho de banda promedio dependiendo únicamente de la probabilidad de pérdida de paquetes (y de variables como el RTT y el MSS).

8.6. Fuentes

- Rodrigo Castro. Teoría de las Comunicaciones, Clase Teórica 6. Primer cuatrimestre, 2020.
- Rodrigo Castro. Teoría de las Comunicaciones, Clase Teórica 8. Primer cuatrimestre, 2020.
- Paula Verghelet. Teoría de las Comunicaciones, Clase Práctica 7. Primer cuatrimestre, 2020.
- Marcos Cervetto. Teoría de las Comunicaciones, Clase Práctica 10. Primer cuatrimestre, 2020.
- Julián Sackmann. Teóricas de Teoría de las Comunicaciones. 2012.
- Guido Tagliavini Ponce. Resumen de Teoría de las Comunicaciones. 2017.

9. Nivel de Aplicación

9.1. Introducción

El nivel de aplicación corresponde a la capa 7 del modelo OSI, o la cuarta del modelo TCP/IP que se utiliza en la práctica (es la última capa en ambos, pero en este último engloba a las tres capas superiores del modelo OSI).

Al igual que con el nivel de transporte, la comunicación en el nivel de aplicación es end-to-end, es decir que **funciona en sistemas finales (hosts)**.

El nivel de aplicación corresponde a la conjunción de **comunicación y procesos** distribuidos. Por ejemplo, servicios de correo electrónico, web, transferencia de archivos, mensajería, etc. Se basa en el **intercambio de mensajes** para la implementación de aplicaciones.

Los **protocolos** de capa de aplicación son una parte de la aplicación, y los mensajes intercambiados por las aplicaciones son acciones ejecutadas por los **procesos en los hosts**, y los **servicios provistos por protocolos de capas inferiores** (e.g. TCP, UDP).

- **Procesos:** programas que se ejecutan en un host. En un mismo host, dos programas se comunican entre sí vía **comunicación interproceso**, mientras que los procesos que se ejecutan entre distintos hosts se comunican con un **protocolo de capa de aplicación**.
- **Agentes de usuario:** interfaces con un usuario *por encima*, y una red *por debajo*. Implementan interfaces de usuario y protocolos a nivel aplicación. Ejemplos de esto son el navegador para web, el lector de correo para correo electrónico, etc.

Los protocolos de capa de aplicación intercambian **mensajes de petición** y de **respuesta**. Cada mensaje tiene su sintaxis, semántica y reglas particulares, sobre las cuales se deben *poner de acuerdo* los hosts que se comunican.

9.2. Paradigma tipo Cliente-Servidor

Una aplicación de red tiene *típicamente* dos partes:

1. Cliente:

- **Inicia** el contacto con el servidor.
- **Solicita** un servicio del servidor.
- Ejemplos: navegador web, lector de correo.

2. Servidor:

- Escucha y **recibe** solicitudes de los clientes.
- **Provee** el servicio solicitado al cliente.
- Ejemplos: el servidor web envía una página, el servidor de correo entrega un correo.

Esto no es siempre así; puede haber casos en que un servidor inicie una comunicación, por ejemplo enviando una notificación masiva a todos sus clientes avisándoles sobre algún cambio de estado (esto no suele ocurrir en Internet, pero sí puede pasar en redes internas de organizaciones).

Comunicación de procesos en la red: El proceso envía/recibe mensajes hacia/desde su **socket** (un socket es como una *puerta*: el proceso emisor envía el mensaje por su puerta, y asume que hay una infraestructura del otro lado que llevará el mensaje a la puerta del proceso receptor).

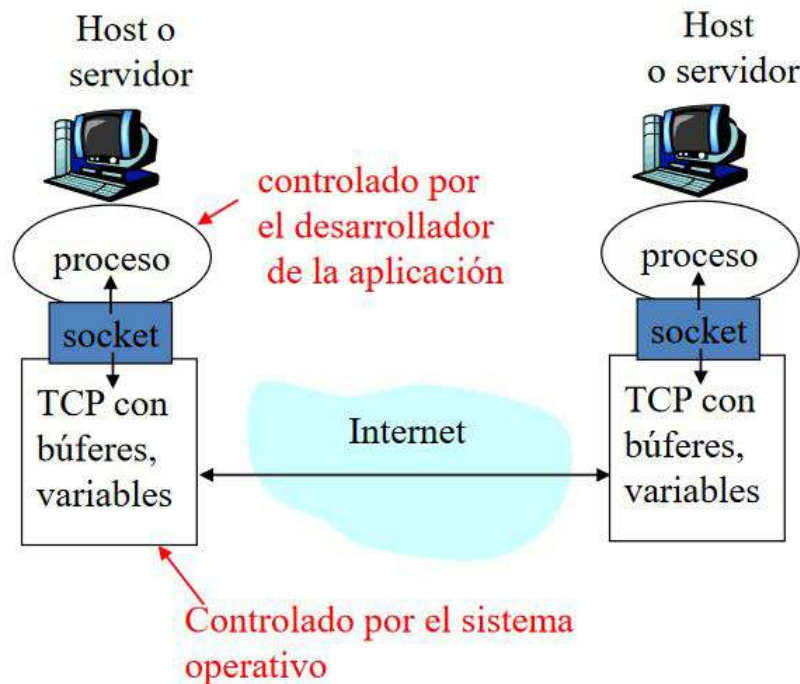


Figura 62: Esquema de una comunicación entre procesos en la red

9.3. Servicios de Transporte

Recordar:

- TCP: orientado a conexión, confiabilidad, control de flujo, control de congestión. No temporización, ni garantía de ancho de banda mínimo.
- UDP: sin conexión, no confiabilidad, ni control de flujo o congestión, ni temporización ni garantía de ancho de banda.

Diferentes aplicaciones pueden requerir distintos servicios de la capa inferior (transporte). **Dependiendo de esas necesidades**, será que se decide el protocolo de transporte a utilizar (e.g. TCP, UDP). Si una responsabilidad no es cumplida por la capa de transporte, se deberá ocupar de ellos la capa de aplicación. Por ejemplo, si la aplicación se *monta* sobre UDP, habrá que ver qué pasa cuando se pierden paquetes.

Aplicaciones	Protocolo de la capa de aplicación	Protocolo de transporte subyacente
Correo electrónico	SMTP [RFC 2821]	TCP
Acceso a terminales remotos	Telnet [RFC 854]	TCP
Web	HTTP [RFC 2616]	TCP
Transferencia de archivos	FTP [RFC 959]	TCP
Flujo de multimedia	HTTP (YouTube, Netflix) Propietario (Real Networks)	TCP o UDP
Telefonía Internet	SIP [RFC 3261], RTP [RFC 3550], Propietario (Skype)	Típicamente UDP A veces TCP

Figura 63: Protocolos de transporte usados por aplicaciones comunes de Internet.

9.3.1. Criterios de Selección

- Pérdida de datos: algunas aplicaciones pueden tolerar algunas pérdidas (audio, vídeo), mientras que otras requieren una transferencia confiable de datos (transferencia de archivos).
- Temporización: algunas aplicaciones requieren un retardo artificial para ser efectivas (telefonía sobre Internet).
- Ancho de banda: algunas aplicaciones requieren un mínimo de ancho de banda para funcionar bien (audio, video), mientras que otras (*aplicaciones flexibles*) hacen uso del ancho de banda que tengan a su disposición (email).

Aplicación	Pérdida de datos	Ancho de banda	Sensible al delay
Transf. de archivos	No pérdida	Flexible	No
Correo electrónico	No pérdida	Flexible	No
Documentos Web	No pérdida	Flexible	No
Audio/vídeo de tiempo real	Tolerante	Audio: 5Kbps-1Mbps Vídeo: 10Kbps-5Mbps	Sí, 100 mseg
Audio/vídeo almacenado	Tolerante	Igual que el anterior	Sí, pocos seg
Juegos interactivos	Tolerante	Pocos Kbps-10Kbps	Sí, 100 mseg
Mensajería instantánea	No pérdida	Flexible	Depende

Figura 64: Requisitos de los servicios de transporte para aplicaciones comunes.

9.4. Domain Name System (DNS)

9.4.1. Fundamentos

El Domain Name System (Sistema de Nombres de Dominio, o DNS) es un sistema de nombres jerárquico y descentralizado para recursos conectados a una red.

Se encarga de organizar máquinas dentro de dominios, y **resuelve** (*traduce*) **nombres** de hosts (e.g. URLs) **en direcciones IP**.

Es un sistema **distribuido** y **escalable**.

9.4.2. Funcionamiento General

1. Un **proceso de aplicación** llama a un procedimiento llamado **resolvedor**, pasándole el nombre a resolver como parámetro.
2. El resolvedor envía un paquete **UDP** a un **servidor de DNS local**.
3. Ese servidor de DNS local **busca el nombre** y retorna la dirección IP al resolvedor.
4. El resolvedor le **devuelve** la dirección IP resuelta al solicitante.
5. Cuando el solicitante tiene la dirección IP, el programa en cuestión puede establecer una conexión TCP con el destino, enviarle paquetes UDP, etc.

9.4.3. El Espacio de Nombres

- En Internet, el espacio de nombres se divide primero en **dominios de nivel superior** (top-level domains, TLD), que abarcan muchos hosts.
- Cada dominio se divide en **subdominios**, que también se dividen (estructura **jerárquica**).
- Un **TLD** es la más alta categoría de los Fully Qualified Domain Name (nombre de dominio completamente calificado, o **FQDN**) que es traducida a direcciones IP por los DNS.

9.4.4. Registros DNS

El contenido de los registros DNS consiste en tuplas $\langle \text{Tipo}, \text{Valor} \rangle$. En la figura 65 se puede ver lo que significa cada tupla que puede estar contenida en un registro DNS. Además de estas tuplas, los registros contienen el nombre de dominio, el TTL (time-to-live en segundos) y la clase.

Estos registros se encuentran en los repositorios de datos de los DNS, y se usan para, dado un nombre de dominio (FQDN) obtener una dirección IP.

Tipo	Significado	Valor
SOA	Inicio de autoridad	Parámetros para esta zona
A	Dirección IP de un <i>host</i>	Entero de 32 bits
MX	Intercambio de correo	Prioridad, dominio dispuesto a aceptar correo electrónico
NS	Servidor de nombres	Nombre de un servidor para este dominio
CNAME	Nombre canónico	Nombre de dominio
PTR	Apuntador	Alias de una dirección IP
HINFO	Descripción del <i>host</i>	CPU y SO en ASCII
TXT	Texto	Texto ASCII no interpretado

Figura 65: Contenido de un registro DNS.

Un registro de tipo SOA (Start Of Authority) indica el nombre de la fuente primaria de información sobre la zona de cobertura del servidor de nombres, la dirección de correo de su administrador, un número de serie y distintos flags y temporizadores.

9.4.5. Consultas DNS

Hay dos formas elementales en las cuales opera DNS: **recursiva** e **iterativa**:

- Una consulta es **recursiva** si el cliente que la realiza le consulta a un servidor de nombres que le va a consultar a otro servidor de nombres (no puede dar una respuesta por sí solo, debe comunicarse con otros servidores de nombres para dar con la dirección IP del dominio solicitado, y devolverla al cliente).
- Una consulta es **iterativa** cuando el servidor de nombres consultado devuelve una respuesta sin consultar a otros servidores de nombres (incluso si no puede dar una respuesta definitiva para la consulta). Por eso, normalmente la primera consulta es recursiva y las consultas subsiguientes, desencadenadas por la primera, son iterativas, como en la figura 66.

Una consulta DNS como la que se muestra en la figura 66 comienza con un cliente que quiere acceder a un servidor localizado en el Departamento de Ciencias de la Computación de la Universidad de Princeton, en Estados Unidos. Para eso, el cliente dispone de la URL correspondiente.

Entonces, el cliente accede a su servidor de nombres local (local name server), que debe resolver el nombre del dominio en una IP. Esta primera consulta, al servidor local, es **recursiva**.

El servidor de nombres local comienza dirigiéndose a un **Root Name Server** (esto es algo que se configura a mano en cada servidor local, y es donde acude en caso de no tener información alguna sobre el nombre de dominio solicitado).

El Root Name Server no tiene por qué conocer le IP que le solicitó el servidor local, pero sí le puede indicar a quién preguntarle luego (baja un nivel en la jerarquía). En el caso del ejemplo, el Root Name Server le da al servidor local la IP del name server de **.edu**.

Luego, el servidor local le solicita mismo de antes, pero esta vez al servidor de nombres de **.edu**. Este, a su vez, le indica la IP del servidor de nombres de la Universidad de Princeton, que es consultado para luego devolver la IP del Departamento de Ciencias de la Computación.

Este último name server sí conoce la IP del nombre de dominio solicitado, así que se la entrega al servidor local (es el fin de la cadena de jerarquías).

Por último, el servidor de nombres local le devuelve al cliente la IP del nombre que solicitó.

Todas estas consultas desde el servidor local a los distintos servidores de nombres de la jerarquía, son consultas **iterativas**.

El cliente sólo tiene que hacer un único paso (consultar a su servidor de nombres local), que es la consulta recursiva. Por su parte, el servidor de nombres local debe completar tantos pasos como sean necesarios para llegar a resolver el nombre que le solicitó el cliente; estas son todas las consultas iterativas. Ambos tipos de consulta son soportados por DNS.

9.4.6. Respuestas DNS

Un servidor de nombres puede dar dos tipos de respuestas a una consulta: **autoritativa** o **no autoritativa**, dependiendo de si el servidor tiene autoridad para responder el dato que le solicitan, o no. Esta autoridad la tendrán aquellos servidores que tengan responsabilidad sobre la zona.

En el ejemplo de la figura 66, la respuesta que da el servidor de nombres local es no autoritativa (dado que fue obtenida a partir de *preguntarle* a otros servidores), mientras que las provistas por los otros servidores son autoritativas (tienen ciertas garantías sobre corrección, actualidad, etc.).

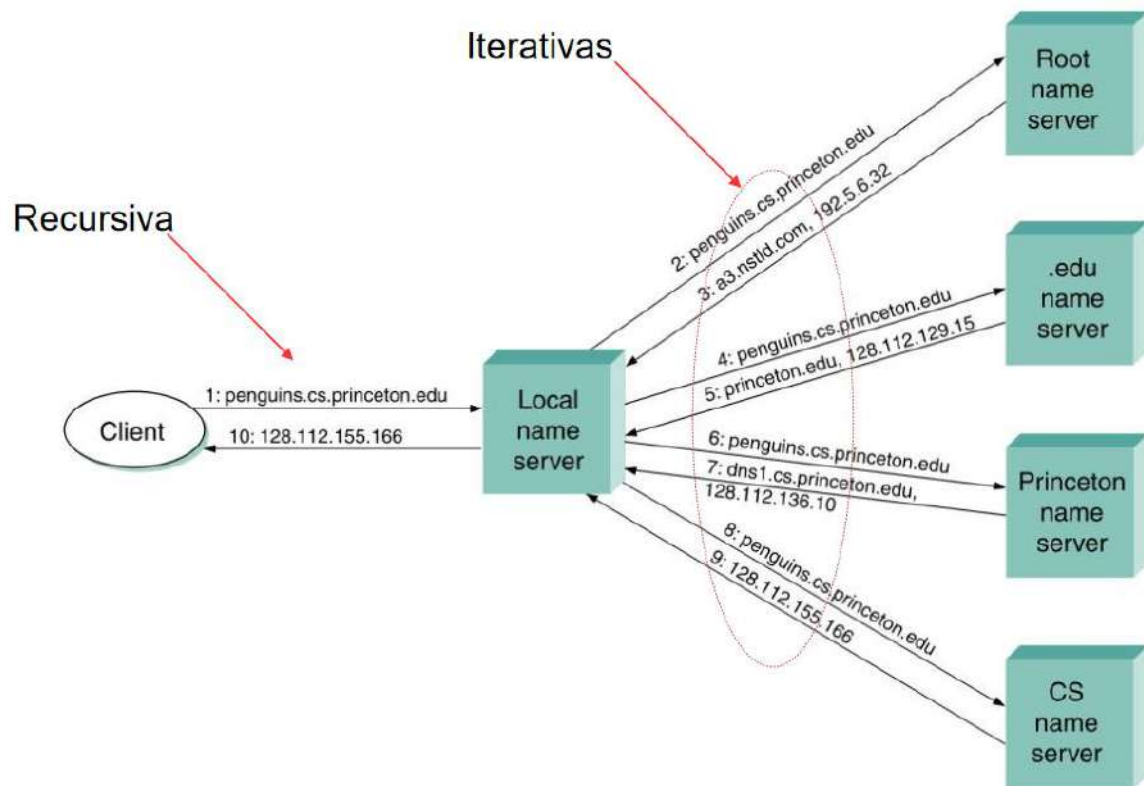


Figura 66: Ejemplo de una secuencia de interacciones para una consulta DNS.

9.4.7. Proceso de una Consulta DNS

1. Alguna **aplicación** necesita resolver un **nombre a una dirección IP**.
2. Se genera un *resolution request* al **resolvedor** del host (es un servicio que corre dentro del host).
3. Este **resolvedor** mira en su **caché** y en alguna lista que pueda tener configurada estáticamente, para ver si puede resolverlo sin hacer una consulta DNS. Si ese nombre existe, se resuelve y devuelve la respuesta. Si no lo tiene va a 4.
4. Como el **resolvedor** no puede resolver el nombre, genera una consulta DNS (principal función del **resolvedor**) al **Local Name Server** (*servidor de nombres local*, que suele ser el servidor DNS que se configura con DHCP). Esta consulta suele ser **recursiva**.
5. El Local Name Server recibe la consulta. Aquí pueden pasar varias cosas:
 - El nombre que se está pidiendo es **local**, con lo cual es posible que esté dentro de la zona del name server y pueda dar una **respuesta autoritativa** y se termina todo ahí.
 - El nombre que se está pidiendo ya fue pedido antes, entonces se encuentra en el **caché** del Local Name Server, con lo cual se devuelve esa **respuesta no autoritativa** (en la

respuesta se indica también cuál es el name server autoritativo que le dio inicialmente esa respuesta).

- El nombre que se está pidiendo **no está en su zona y no se encuentra en caché**. En ese caso, el Local Name Server, auspiciado por el cliente DNS (*resolvedor*) y ejecuta una consulta que suele ser **iterativa**. La consulta se realiza al **Root Name Server**. La respuesta del Root Name Server suele ser: *yo no sé pero preguntale a este name server que te paso*. Ese name server suele ser un TLD server. Y se sigue iterativamente la consulta hasta llegar al name server que tiene la respuesta.
6. El local name server tiene la IP deseada y **devuelve la respuesta**. La respuesta es además **cacheada** por el local name server y muy probablemente también cachee cada uno de los name servers que sirvieron para resolver la consulta iterativa.
 7. El servicio *resolvedor* del host obtiene su respuesta, se la entrega a la aplicación y **cachea** el resultado.

9.5. Correo Electrónico

9.5.1. Componentes Principales

Una aplicación de correo electrónico se compone de tres partes:

1. Agentes de usuario:
 - También conocidos como *lectores de correo*.
 - Sirven para componer, editar y leer correos.
 - Entrada y salida de mensajes almacenados en el servidor de correo.
 - Ejemplos: Microsoft Outlook, Mozilla Thunderbird.
2. Servidores de correo:
 - Buzón de correo: contiene los mensajes de entrada del usuario.
 - Cola de mensajes: contiene los mensajes de correo de salida (encolados para ser enviados).
3. Protocolos de transferencia de correos:
 - Sirven para intercambiar mensajes de correo electrónico entre servidores.
 - El cliente envía correo al servidor, y el servidor recibe correo de otro servidor.
 - Ejemplo: SMTP.

9.5.2. Simple Mail Transfer Protocol (SMTP)

Simple Mail Transfer Protocol (*Protocolo para Transferencia Simple de Correo*, o SMTP) es un protocolo de comunicación para **transferencia de correos electrónicos**.

Características:

- Al tratarse de correo electrónico, no hay requerimientos de temporización ni de ancho de banda mínimo, pero sí es importante que no haya pérdida de paquetes.

- Utiliza **TCP** para transferir **confiablemente** los mensajes de correo del cliente al servidor, usando el puerto 25.
- **Transferencia directa** desde el servidor que envía hasta el servidor que recibe.
- Una transferencia tiene tres fases:
 1. Acuerdo (saludo).
 2. Transferencia de mensajes.
 3. Cierre.
- Los comandos se realizan en texto plano ASCII, y las respuestas constan de códigos de status y frases.

Ejemplo de mensaje: A continuación se muestra la secuencia de pasos que se llevan a cabo cuando Alice le envía un mensaje a Bob (ver figura 67):

1. Alice utiliza su agente de usuario para escribir un mensaje a la dirección de Bob.
2. El agente de usuario de Alice envía un mensaje a su servidor de correo, y el mensaje se coloca en la cola.
3. El lado cliente de SMTP abre una conexión TCP con el servidor de correo de Bob (para esto probablemente tenga que pasar por DNS).
4. El cliente SMTP envía el mensaje de Alice usando la conexión TCP.
5. El servidor de correo de Bob deposita el mensaje en el buzón de correo de Bob.
6. Bob usa su agente de usuario para abrir y leer el mensaje.

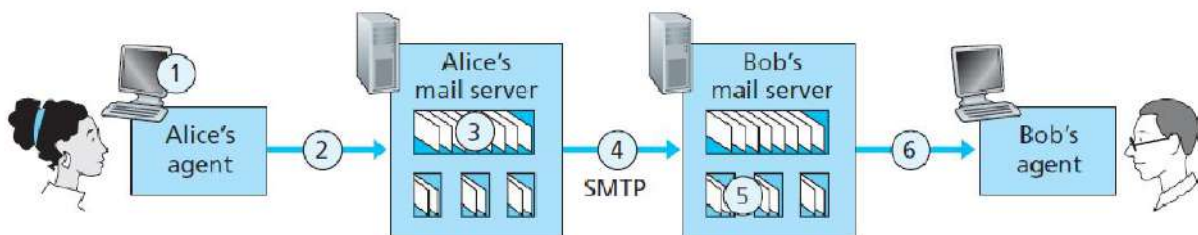


Figura 67: Alice envía un mensaje a Bob.


```

C: HELO crepes.fr
S: 250 Hello crepes.fr, pleased to meet you
C: MAIL FROM: <alice@crepes.fr>
S: 250 alice@crepes.fr ... Sender ok
C: RCPT TO: <bob@hamburger.edu>
S: 250 bob@hamburger.edu ... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: Do you like ketchup?
C: How about pickles?
C: .
S: 250 Message accepted for delivery
C: QUIT
S: 221 hamburger.edu closing connection

```

Figura 68: Ejemplo de interacción SMTP.

9.5.3. Multipurpose Internet Mail Extensions (MIME)

Multipurpose Internet Mail Extensions (*extensiones multipropósito de correo de internet*, o MIME) es un estándar de Internet que extiende el formato para los correos electrónicos, de manera tal que estos puedan soportar más que texto plano ASCII. Por ejemplo, soporta otros tipos de caracteres y archivos adjuntos (audio, vídeo, imágenes, etc.).

En el encabezado del mensaje, se declara el tipo de contenido MIME, y esta información es utilizada del lado del receptor para mostrar la información adecuadamente.

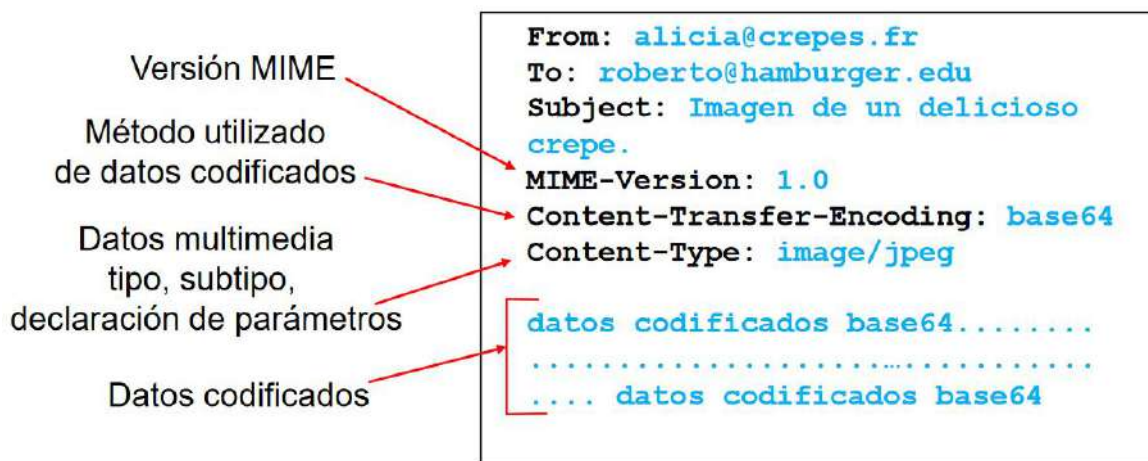


Figura 69: Formato de un correo que usa MIME.

9.5.4. Transferencia y Entrega de Mensajes

Además de SMTP, existen protocolos que sirven para realizar la entrega de mensajes al agente de usuario de un receptor, es decir para **descargar los mails de la casilla desde el servidor de correo**. Por ejemplo, como se ve en la parte b de la figura 70, el host emisor envía un correo electrónico utilizando SMTP, y este llega al buzón del agente de transferencia de mensajes (ya que por ejemplo, podría ser que un usuario utilice el servicio de correo del ISP). Luego, ese punto intermedio entre el emisor y el receptor tiene un servidor POP3, desde el cual se envía el mensaje a la computadora del usuario, donde este puede descargarlo.

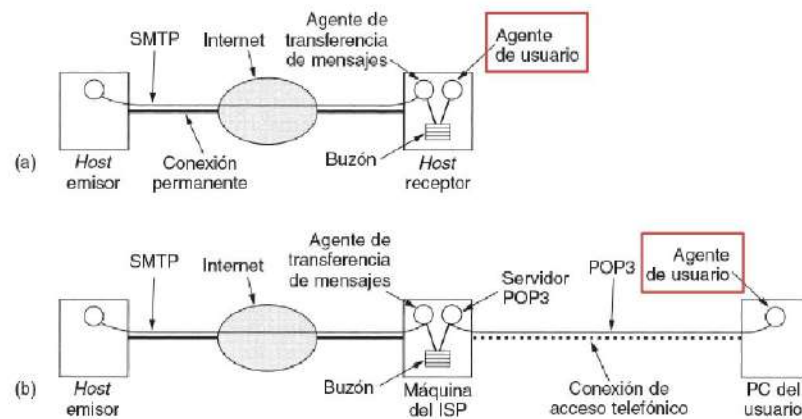


Figura 70: En a, se puede ver una transferencia de mensajes *directa* entre emisor y receptor. En b, se muestra cómo puede participar también un agente de transferencia de mensajes que utiliza protocolos como POP3.

Característica	POP3	IMAP
En dónde se define el protocolo	RFC 1939	RFC 2060
Puerto TCP utilizado	110	143
En dónde se almacena el correo electrónico	PC del usuario	Servidor
En dónde se lee el correo electrónico	Sin conexión	En línea
Tiempo de conexión requerido	Poco	Mucho
Uso de recursos del servidor	Mínimo	Amplio
Múltiples buzones	No	Sí
Quién respalda los buzones	Usuario	ISP
Bueno para los usuarios móviles	No	Sí
Control del usuario sobre la descarga	Poco	Mucho
Descargas parciales de mensajes	No	Sí
¿Es un problema el espacio en disco?	No	Con el tiempo podría serlo
Sencillo de implementar	Sí	No
Soporte amplio	Sí	En crecimiento

Figura 71: Comparación entre POP3 e IMAP.

Existen distintos protocolos para la entrega/obtención de correo electrónico. Por ejemplo:

- Post Office Protocol 3 (*protocolo de oficina de correo 3*, o **POP3**). Corre sobre TCP, y se dice que está **centrado en el cliente**:
 - Los correos se descargan del servidor (y son eliminados de allí) y se administran localmente.
 - Como toda la información se descarga, no hace falta conexión para acceder a los correos.
 - El backup pasa a ser responsabilidad del cliente.
 - El servidor sólo tiene una bandeja de entrada de la que el cliente descarga los mensajes nuevos. La administración de mails (por ejemplo, la separación en carpetas) es responsabilidad del cliente.

- Internet Message Access Protocol (*protocolo de acceso a mensajes de Internet*, o **IMAP**). También corre sobre TCP, pero se dice que está **centrado en el servidor**:
 - Todos los mensajes quedan almacenados en el servidor. Cuando se desea acceder a la casilla de correo, debe descargarse la información desde el servidor.
 - Es útil cuando se accede a la casilla desde distintos hosts.

9.6. Web y HyperText Transfer Protocol (HTTP)

9.6.1. Fundamentos

HyperText Transfer Protocol (*protocolo de transferencia de hipertexto*, o **HTTP**) es un protocolo de aplicación que permite la **transferencia de información en la Web**.

Características generales:

- Una **página web** consta de objetos, y está formada por un **archivo HTML base**, que incluye objetos referenciados en ella.
- Un **objeto** puede ser un archivo HTML, una imagen JPEG, un archivo de audio, etc.
- Cada objeto es direccionable por una URL (Uniform Resource Locator, o *localizador de recursos uniforme*).
- Una URL está compuesta por el nombre del host, y el nombre de ruta.
- El nombre del host es resuelto vía DNS para obtener una dirección IP. Luego, el navegador de Internet, para ir a buscar algo a esa dirección, deberá agregarle algo para comunicarse con la aplicación correcta del servidor hosteado allí: el **puerto** correspondiente (que suele ser el 80 para Web).
- Una vez obtenida la IP y el puerto (es decir, el socket), se llega a una especie de directorio, con objetos y directorios. Entonces, el browser debe obtener el objeto dado por el nombre de ruta.

HTTP sigue el modelo **cliente-servidor**:

- El cliente es el **navegador**, que solicita, recibe y descarga objetos Web.
- El servidor es el **servidor Web** que devuelve los objetos solicitados en respuesta a las peticiones del cliente.

9.6.2. Operación

Usos de TCP:

1. El cliente inicia la conexión TCP al servidor en puerto 80.
2. El servidor acepta la conexión TCP del cliente.
3. Los mensajes HTTP son intercambiados entre el cliente y el servidor.
4. La conexión TCP se cierra.



Figura 72: Ejemplo de interacción cliente-servidor en HTTP.

El servidor no conserva ninguna información sobre las peticiones previas de clientes (**HTTP opera *sin estado***). Los protocolos que sí conservan estado son más complicados (por ejemplo, el uso de cookies).

Abrir y cerrar conexiones TCP por cada solicitud no es muy eficiente, y es una de las cosas que contempla la versión 1.1 de HTTP.

9.6.3. Mensajes HTTP

Hay dos tipos de mensajes HTTP:

- De **petición**. Algunos métodos disponibles son:
 - **GET**: solicita que el recurso transfiera una representación de su estado, sin ningún efecto secundario.
 - **POST**: solicita que el recurso procese la representación incluida en la solicitud (por ejemplo, para llenar formularios).
 - **HEAD**: pide al servidor que excluya el objeto solicitado, es decir que envíe solo la cabecera de la respuesta.
 - **PUT**: descarga el archivo especificado en el cuerpo de entidad a la ruta especificada en el campo URL.
 - **DELETE**: borra el archivo especificado en el campo URL.

Diagram illustrating an example of an HTTP request message. The message is shown in blue text, and the annotations are in blue text with red arrows pointing to the corresponding parts of the message.

```

GET /dir/pagina.html HTTP/1.1
Host: www.escuela.edu
User-agent: Mozilla/4.0
Connection: close
Accept-language: fr
  
```

Annotations:

- Línea de petición (GET, POST, comandos HEAD)**: Points to the first line of the message.
- Líneas de cabecera**: Points to the header lines (Host, User-agent, Connection, Accept-language).
- Retorno de carro y avance de línea que indican el final del mensaje**: Points to the end of the message, indicating the return of the carriage and the advance of the line.
- (Retorno de carro extra, avance de línea)**: Points to the end of the message, indicating the return of the carriage and the advance of the line.

Figura 73: Ejemplo de mensaje de petición.

- De **respuesta**. Contiene un código de status y una frase, además de una cabecera con información (fecha, servidor, tipo de contenido, etc.) y los datos solicitados. Algunos códigos típicos de respuesta son:
 - 200 OK: petición exitosa, el objeto solicitado aparece en el mensaje.
 - 301 Moved Permanently: el objeto solicitado ha sido movido, y se especifica su nueva localización en el mensaje.
 - 400 Bad Request: el servidor no comprendió el mensaje de petición.
 - 404 Not Found: el documento pedido no existe en el servidor.
 - 500 HTTP Version Not Supported: la versión de HTTP indicada en el mensaje de petición no es soportada por el servidor.

Diagram illustrating an example of an HTTP response message. The message is shown in green text, and the annotations are in blue text with red arrows pointing to the corresponding parts of the message.

```

HTTP/1.1 200 OK
Connection: close
Date: Thu, 06 Aug 1998 12:00:15 GMT
Server: Apache/1.3.0 (Unix)
Last-Modified: Mon, 22 Jun 1998 ...
Content-Length: 6821
Content-Type: text/html
datos datos datos datos datos ...
  
```

Annotations:

- Línea de status (protocolo del código de estatus y la frase de estatus)**: Points to the first line of the message.
- Líneas de cabecera**: Points to the header lines (Connection, Date, Server, Last-Modified, Content-Length, Content-Type).
- Datos, por ejemplo, el archivo HTML solicitado**: Points to the body of the message (datos datos datos datos datos ...).

Figura 74: Ejemplo de mensaje de respuesta.

9.6.4. Tiempo de Respuesta

Definición de RTT: tiempo necesario para enviar un paquete pequeño desde el cliente hasta el servidor y después de vuelta al cliente.

Tiempo de respuesta para una transacción HTTP:

- Un RTT para iniciar la conexión TCP.
- Un RTT para la petición HTTP y los primeros bytes de respuesta HTTP de vuelta.
- Tiempo de transmisión del archivo: $2 \times \text{RTT} + \text{tiempo de transmisión}$.

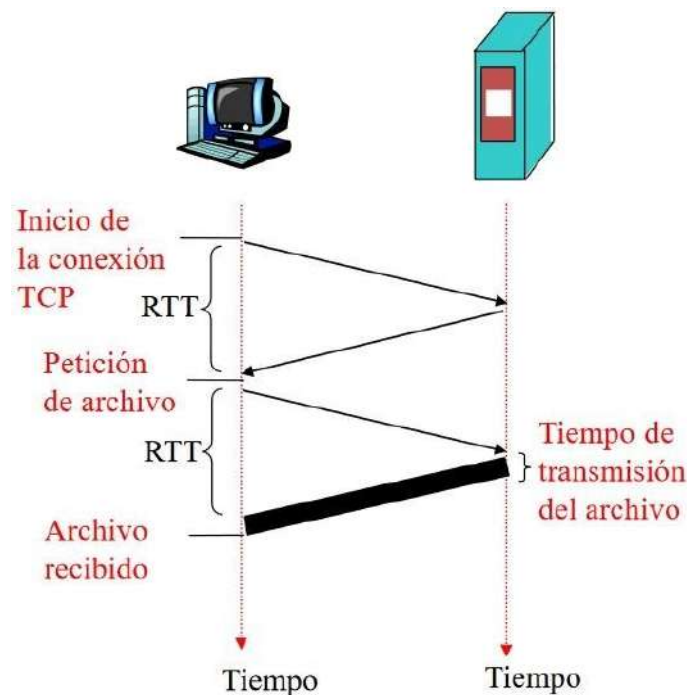


Figura 75: Paso del tiempo en una transferencia de HTTP.

9.6.5. Conexiones HTTP

- Conexiones **no persistentes** (HTTP 1.0): se envía un objeto como máximo por cada conexión TCP.
 - Requieren dos RTTs por cada objeto solicitado.
 - El sistema operativo debe asignar los recursos del host para cada conexión TCP.
 - Los navegadores suelen abrir conexiones TCP paralelas para traer los objetos referenciados.
- Conexiones **persistentes** (HTTP 1.1): se puede enviar múltiples objetos con una única conexión TCP entre el cliente y el servidor.
 - El servidor deja la conexión abierta tras enviar la respuesta.
 - Los mensajes HTTP posteriores entre el mismo par cliente-servidor se envían usando la misma conexión.

- Conexiones persistentes sin entubamiento:
 - El cliente sólo emite una nueva petición una vez que ha recibido la respuesta anterior.
 - Un RTT por cada objeto referenciado.
- Conexiones persistentes con entubamiento:
 - El cliente hace su petición tan pronto como encuentra un objeto referenciado.
 - Un RTT para todos los objetos referenciados.
 - Es la opción por defecto en HTTP 1.1.

9.6.6. Cookies: Mantenimiento del Estado

Actualmente la mayoría de sitios web importantes utilizan **cookies**. Estas tienen cuatro componentes:

1. Línea de **cabecera** del cookie en el mensaje HTTP de **respuesta**.
2. Línea de **cabecera** de cookie en el mensaje HTTP de **petición**.
3. **Archivo** de cookie, que se almacena **en el host del usuario** y es gestionado por su navegador.
4. **Base de datos** de respaldo en el sitio **Web**.

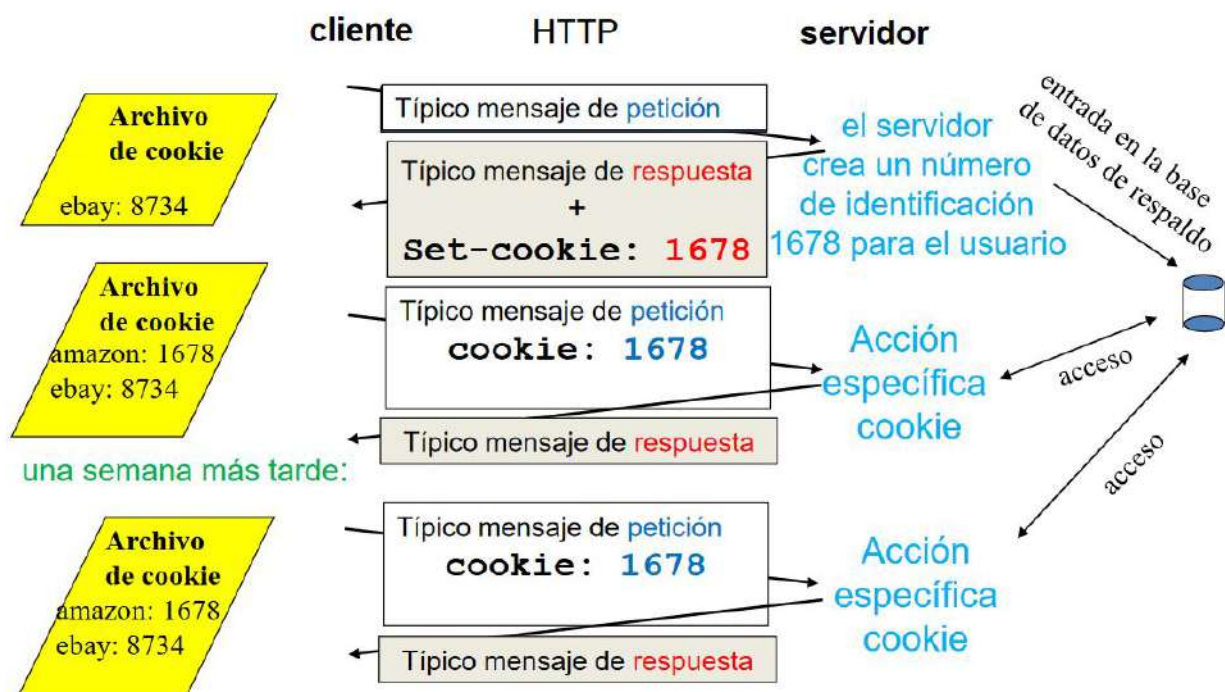


Figura 76: Esquema del uso de cookies para mantener el estado.

9.7. Peer-to-Peer y BitTorrent

- BitTorrent es un protocolo para distribución de archivos en Internet.
- Los nodos que descargan el archivo contribuyen a distribuirlo, subiendo partes del archivo a otros nodos que lo están bajando.
- Posibilita que un nodo con poco *ancho de banda* distribuya un archivo a muchos otros, sin sufrir un incremento lineal en la demanda a medida que crece el número de nodos que está descargando el archivo.
- El conjunto de nodos (*peers*) descargando el archivo se denomina *swarm*.
- El archivo a bajar (replicar) se divide en piezas de tamaño fijo para facilitar su intercambio entre peers.
- Cuando un peer termina de replicar una pieza entera, puede empezar a ofrecerla a los demás peers.
- Así es como las piezas y el archivo se replican una vez descargadas del nodo que tenía el archivo originalmente (primer *seed*).

9.8. Fuentes

- Rodrigo Castro. Teoría de las Comunicaciones, Clase Teórica 7. Primer cuatrimestre, 2020.
- Esteban Lanzarotti. Teoría de las Comunicaciones, Clase Práctica 8. Primer cuatrimestre, 2020.
- Leonardo Balbiani. Teoría de las Comunicaciones, Clase Práctica 9. Primer cuatrimestre, 2020.
- Guido Tagliavini Ponce. Resumen de Teoría de las Comunicaciones. 2017.

10. Seguridad en Redes

10.1. Conceptos

Cada tecnología de seguridad informática puede ocuparse de un subconjunto de las siguientes propiedades:

- **Confidencialidad:** los mensajes sólo deben poder **ser entendidos** por las partes especificadas en la comunicación.
- **Integridad:** los mensajes enviados **no pueden ser modificados** durante su transmisión. Por ejemplo, si se le pide algo al servidor de Google, uno no quiere que nadie modifique el contenido de `www.google.com` mientras viaja por Internet hasta la computadora del usuario.
 - Originalidad: el mensaje no es una **copia artificial** repetida.
 - Temporalidad: el mensaje no fue **demorado** maliciosamente.
- **Autenticación:** ninguna parte puede **asumir la identidad** de otra parte sin autorización, es decir que cada parte es quien dice ser. Por ejemplo, si se le solicita al servidor web de Google que devuelva el contenido de `www.google.com`, no es deseable que alguien asuma la identidad de Google y devuelva otra cosa.
- **Autorización** (control de acceso): sólo ciertos usuarios remotos pueden realizar ciertas acciones permitidas.
- **Disponibilidad:** todo usuario potencial tendrá su oportunidad de ser considerado, y eventualmente admitido.
- **No repudiación:** ninguna de las partes puede **negar haber participado** en una transacción.

Cada una de estas propiedades puede ser atacada con ciertos mecanismos, con el objetivo de ser vulnerada.

10.2. Protocolos y Capas

Todas las características mencionadas antes se pueden atender en distintas capas:

Por ejemplo, en el nivel de red, existe un conjunto de protocolos llamado **IPsec**, que utiliza un Authentication Header para proporcionar integridad, autenticación y no repudio; y un Encapsulating Security Payload para garantizar confidencialidad. Agregarle estas funcionalidades a los paquetes IP implica que debe existir compatibilidad con lo anterior (mantener la estructura original, y agregarle cosas al encabezado).

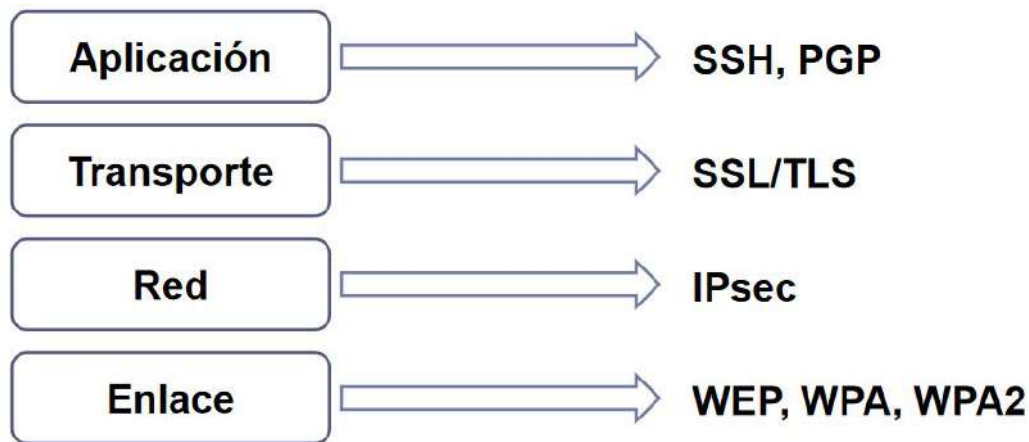


Figura 77: La seguridad se implementa para resolver problemas **en diferentes capas** de la arquitectura de una red.

10.3. Introducción a la Criptografía

10.3.1. Conceptos

- **Cifrado**: transformación **carácter por carácter** (o bit por bit) sin importar la estructura lingüística del mensaje.
- **Código**: reemplazar **una palabra con otra** palabra o símbolo.
- **Plaintext** (*texto plano*): el mensaje a ser encriptado. Son tomados como una **entrada**, y **transformados** por una **función parametrizada con una clave**.
- **Ciphertext** (*texto encriptado*): **salida** del proceso de encripción (cifrado).
- **Intruso pasivo**: puede escuchar y copiar el ciphertext completo.
- **Intruso activo**: puede modificar el ciphertext.
- **Criptoanálisis**: técnica de descifrar mensajes.
- **Criptografía**: técnica de crear ciphers.
- **Criptología**: formada por la criptografía y el criptoanálisis.

10.3.2. Métodos Básicos de Cifrado

Sustitución: Consiste en reemplazar cada letra o grupo de letras por otra letra o grupo de letras, para **enmascarar** el texto original, **preservando el orden** (e.g. cifrado del César, que consiste en rotar el alfabeto en una cantidad fija de letras; sustitución mono alfabética, donde cada letra se mapea a otra cualquiera).

Este tipo de cifrado es fácil de quebrar gracias a las propiedades estadísticas propias de los lenguajes naturales (en los que están escritos los plaintexts).

Transposición: Se **reordenan** las letras, pero no se *enmascaran*. La clave de este método es una palabra sin letras repetidas, siendo su propósito numerar por *columnas* el alfabeto.

10.3.3. Principio de Kerckhoff

*Todos los **algoritmos** deben ser **públicos**, sólo las **claves** deben ser **secretas**.*

La idea de esto es que mantener en secreto el algoritmo no es eficaz (*seguridad por desconocimiento*). Este principio es usado por la mayoría de sistemas de la criptografía moderna.

Teniendo en cuenta que la clave será el secreto, es crítico determinar su **longitud** en el proceso de diseño. A mayor longitud de clave, mayor **factor de trabajo** (tiempo requerido para romper la clave). Este factor de trabajo **crece exponencialmente con el tamaño de la clave**, por lo que el objetivo es dar con un **algoritmo robusto** y una **clave suficientemente larga** como para que sea impráctico intentar romperla.

10.3.4. One-Time Pads

El One-Time Pad (*relleno de un solo uso*) es una técnica de cifrado **inviolable**, pero que requiere una clave de único uso, de longitud no menor a la del plaintext a cifrar. Se utiliza la compuerta lógica XOR para encriptar y desencriptar usando la clave.

El ciphertext resultante es imposible de romper siempre que se cumplan ciertas condiciones (clave aleatoria, suficientemente larga, no reutilizada y secreta), pero a pesar de cumplir con el principio de Kerckhoff, es un método **poco práctico**, ya que distribuir y mantener en secreto la clave es difícil.

10.3.5. Cifrado de Bloque Iterativo

Dada una entrada, se la divide en bloques y se aplican transformaciones a cada parte. Luego, se juntan los resultados y transformarlos en conjunto para obtener una salida. A su vez, dicha salida es entrada para una nueva iteración del mismo método, y así sucesivamente.

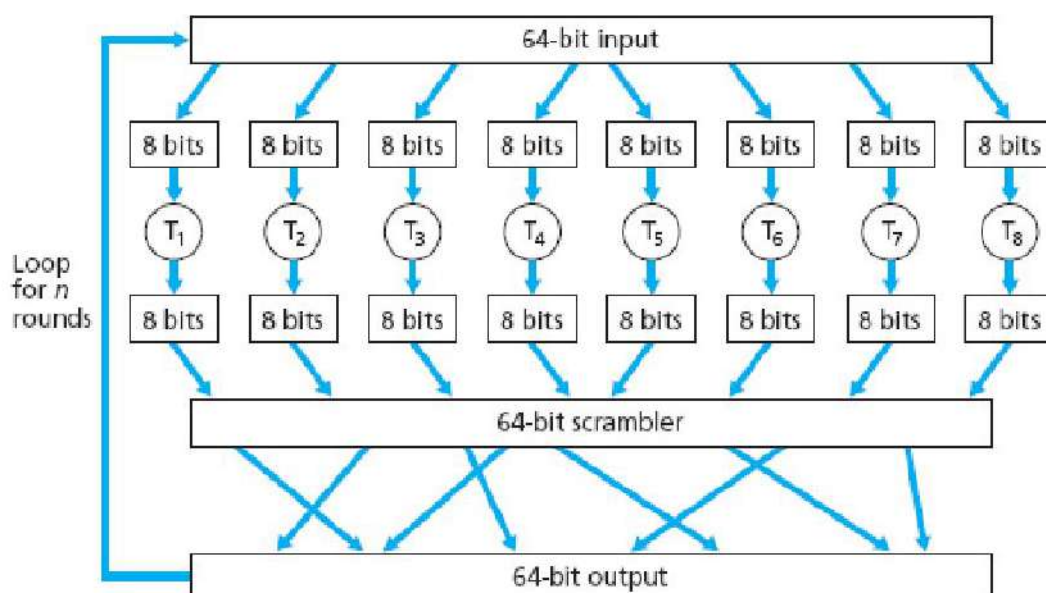


Figura 78: Esquema de un método de cifrado iterativo por bloques.

10.3.6. Data Encryption Standard (DES)

- El Data Encryption Standard (IBM, 1977) es un algoritmo de cifrado muy influyente en el avance de la criptografía.
- Se basa en el **cifrado de bloques**, es **orientado a bits**, y utiliza tanto **sustitución** como **transposición**.
- Trabaja en bloques de a 64 bits, donde cada bloque ejecuta 16 iteraciones con distintas claves.
- Fue quedando obsoleto por el largo de la clave y los avances computacionales que agilizaron los ataques por fuerza bruta.
- 3DES fue una variante de mayor robustez del algoritmo, que consiste en armar una *cascada* de aplicaciones de DES: primero encriptar con una clave, después desencriptar con otra clave, y por último encriptar una vez más con una clave distinta.

10.4. Criptografía de Clave Simétrica

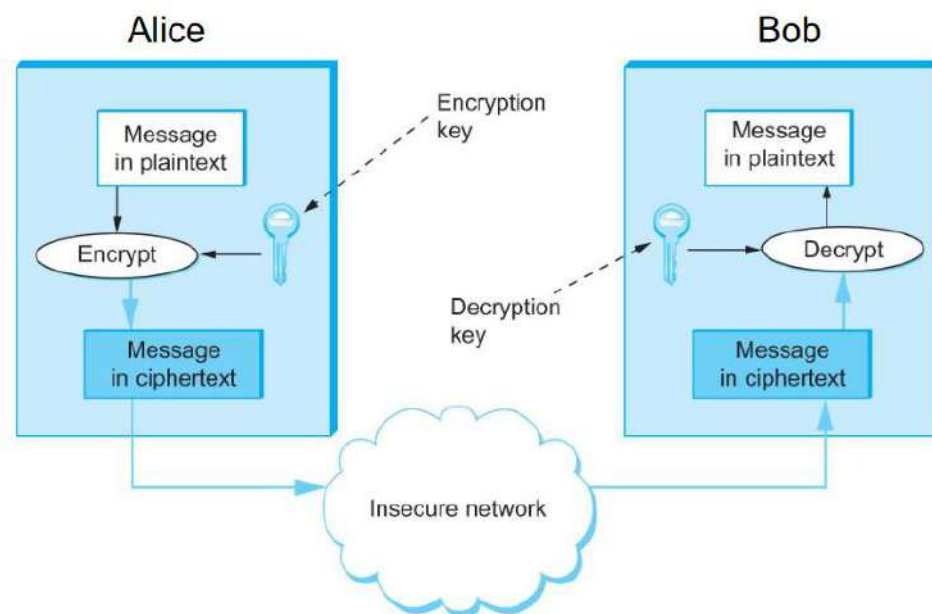


Figura 79: Alice le envía un mensaje a Bob, y ese mensaje es encriptado y desencriptado con la misma clave.

Los puntos fundamentales de la figura 79 son:

- La red por la que viajará el mensaje de Alice hacia Bob es **insegura**: los participantes de la comunicación quieren que los intrusos no sean capaces de leer el mensaje.
- El algoritmo de encriptación y el algoritmo de desencriptación son distintos pero están relacionados, y ambos están parametrizados por una clave (key).
- La clave de encriptación y desencriptación es la misma (por eso se habla de *simétrica*).

En la figura 80, se puede ver el modelo de encriptación para un sistema de criptografía simétrica.

- Se usa la notación $C = E_K(P)$ para representar la encriptación del plaintext P usando la clave K con el método de encriptación E , que genera el ciphertext C .
- $P = D_K(C)$ representa la descriptación de C usando la clave K (igual que la de encriptación), para obtener el plaintext P .
- Es decir, $P = D_K(E_K(P))$.

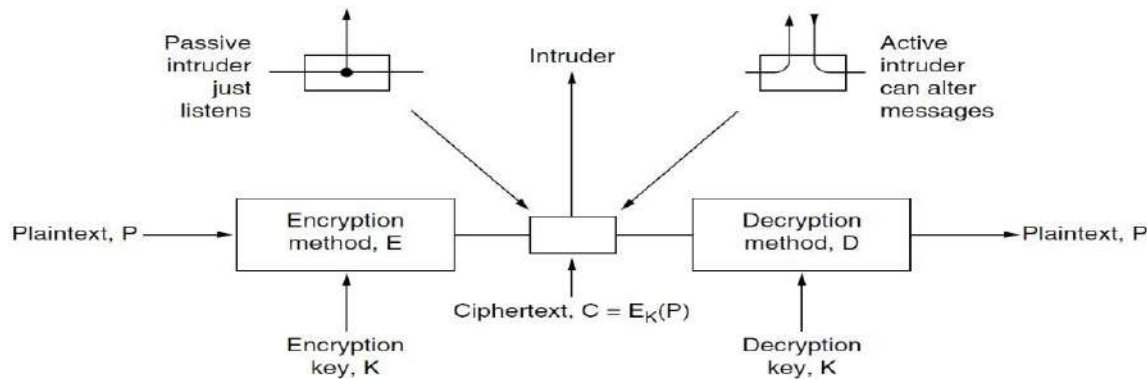


Figura 80: Modelo genérico para clave simétrica.

Los modelos de clave simétrica requieren que el emisor y el receptor conozcan la **clave secreta compartida**. Esto puede ser un problema para ponerse de acuerdo en qué clave utilizar, y en cómo distribuirla manteniendo el secreto.

10.5. Criptografía de Clave Asimétrica

- El emisor y el receptor **no comparten** una **clave secreta**.
- Clave de **encriptación pública**, conocida por todos.
- Clave de **desencriptación privada**, conocida sólo por el receptor.
- Las claves de encriptación y desencriptación deben ser lo suficientemente diferentes como para que no se pueda calcular la segunda a partir de la primera.
- Requisitos:
 - Debe ser computacionalmente fácil, teniendo en cuenta el conexto de uso, cifrar o descifrar (teniendo la clave correcta).
 - Debe ser computacionalmente inviable derivar la clave privada a partir de la clave pública o un texto descifrado.

Este mismo concepto se puede usar a la inversa para asegurar **autenticación**: el emisor encripta su mensaje usando su clave privada, y el receptor lo desencripta con la clave pública, conocida, del emisor. De esta manera, el receptor puede estar seguro de que el mensaje viene efectivamente del emisor.

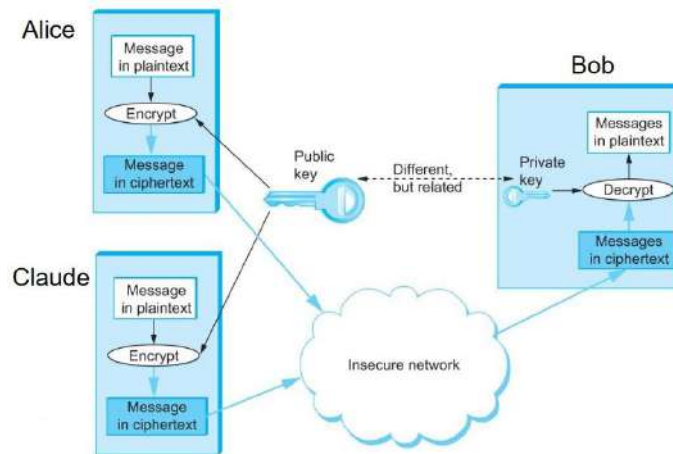


Figura 81: Alice y Claude le envían mensajes a Bob mediante una red insegura, usando la misma clave pública de Bob, K_B^+ , para encriptar el plaintext. Bob utiliza su clave privada K_B^- (relacionada con la pública, pero diferente) para desencriptar los ciphertexts.

10.5.1. Algoritmo de Rivest-Shamir-Adleman (RSA)

El algoritmo RSA es capaz de producir un **par de claves** (pública y privada) que, debido a ciertas propiedades matemáticas acerca de los números primos, módulos y exponenciación, **no son derivables una de la otra**, pero cumplen la propiedad de ser **inversas**.

Procedimiento para elegir las claves:

1. Elegir dos números primos *grandes*, p y q .
2. Calcular $n = p \times q$, y $z = (p - 1) \times (q - 1)$.
3. Elegir $e < n$ *pequeño* tal que e no tenga factores comunes con z (coprimos).
4. Encontrar un número d (este será secreto), tal que $e \times d - 1$ sea divisible por z (es decir, $r_z(e \times d) = 1$).
5. La clave pública es $K^+ = (n, e)$, y la clave privada es $K^- = (n, d)$.

Dados los (n, e) y (n, d) calculados anteriormente:

- Para **encriptar** un plaintext m , calcular $c = r_n(m^e)$ (el resto de dividir m^e por n).
- Para **desencriptar** un ciphertext c , calcular $m = r_n(c^d)$ (el resto de dividir c^d por n).

Es decir:

$$m = r_n(c^d) \quad (1)$$

donde $c = r_n(m^e)$.

Una **propiedad** importante que vale en RSA es:

$$K^-(K^+(m)) = m = K^+(K^-(m)) \quad (2)$$

Es decir, usar primero la clave pública y después la privada es equivalente a usar primero la privada y luego la pública. Esto es útil para firma digital.

Eficiencia: A mismo largo de clave, RSA es más costoso computacionalmente (más que nada por las exponenciaciones) que alternativas como DES o AES (evolución de DES). Es por esto que a veces se utilizan mecanismos de encriptación **híbridos**: primero se usa RSA para acordar una clave, y luego usar un mecanismo como AES por ser una opción menos demandante.

10.6. Firma Digital

Es un mecanismo que asegura autenticidad, integridad y no repudiación. La idea de la firma digital es que el receptor de un mensaje pueda estar seguro de que dicho mensaje viene efectivamente del emisor esperado.

Para esto, el emisor, que conoce su clave privada, la usa para *firmar* el documento. Luego, como con cualquier mensaje en un contexto de clave asimétrica, encripta el mensaje aplicando la clave pública del receptor.

El mensaje viaja por la red, y del lado del receptor, éste lo desencripta usando su clave privada (que sólo él conoce), para luego utilizar la clave pública del emisor, de manera tal que pueda corroborar que fue efectivamente quien lo envió.

Esto funciona gracias a la **conmutatividad** de la aplicación de las claves públicas y privadas vista en el algoritmo RSA.

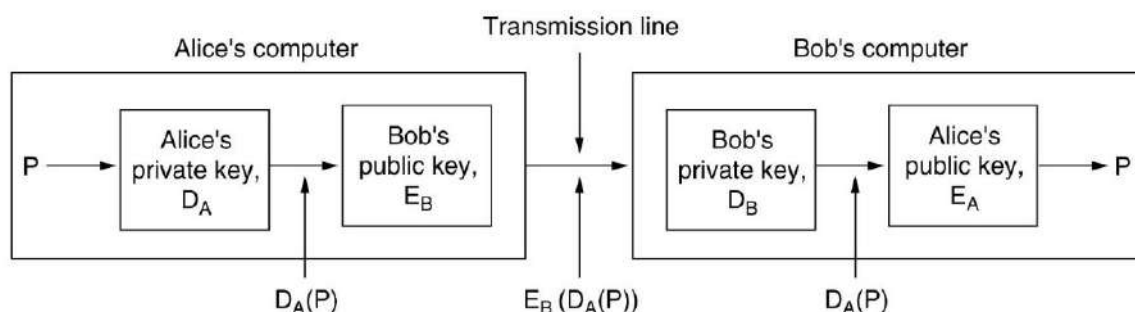


Figura 82: Alice entrega un mensaje firmado digitalmente a Bob. Del lado de Alice, aplica su clave privada al mensaje P , luego lo encripta con la clave pública de Bob. Del lado de Bob, desencripta lo recibido con su clave privada, y luego con la clave pública de Alice, para verificar la firma.

10.6.1. Message Digest

En la práctica, encriptar un mensaje completo usando una clave pública es computacionalmente **costoso** cuando el mensaje es largo. Es por eso que se pone como objetivo tener un mensaje de longitud fija no demasiado grande, para que sea más fácil de computar.

Para esto, se utiliza la **función de dispersión/hash** H , que se le aplica al mensaje m para obtener un **message digest** (*resumen de mensaje*) de **tamaño fijo**, $H(m)$.

Propiedades de una función de hash:

- Es de muchos a uno (puede haber más de un mensaje que tenga el mismo hash).
- Produce un resumen de mensaje de tamaño fijo.
- Dado el resumen de mensaje x , es computacionalmente inviable hallar m tal que $x = H(m)$.

Existen diferentes algoritmos para la función de dispersión, algunos de los cuales son:

- HAVAL.

- MD5.
- SHA-1.
- SHA-2.
- SHA-3.
- BLAKE2.

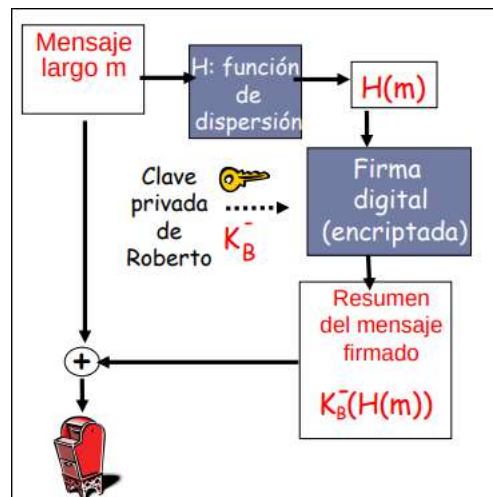


Figura 83: Roberto le envía un mensaje largo a Alicia. Primero, le aplica la función de dispersión, y luego encripta el resultado (el message digest) con su clave privada, para firmarlo.

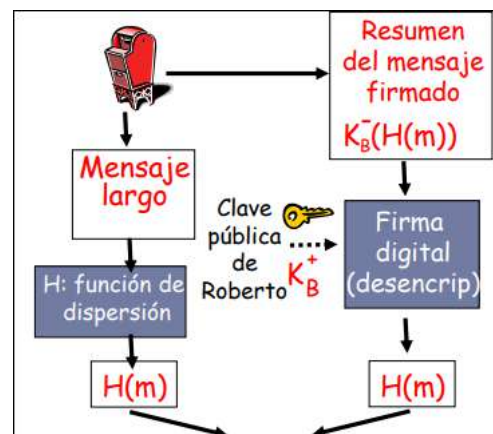


Figura 84: Alicia verifica la firma de Roberto usando la clave pública de éste. Si Alicia le aplica la función de dispersión al mensaje largo de Roberto, debería llegar al mismo resultado que obtuvo desenscriptando el message digest con la clave de Roberto.

10.7. Autenticación

Al iniciar un proceso de comunicación entre dos (o más) partes, la **autenticación** es un mecanismo por el cual las partes involucradas (a partir de ese momento), operen bajo la asunción de que ambas son quien dicen ser.

10.7.1. Autenticación de Dos Vías

En particular, la **autenticación de dos vías** se refiere al proceso por el cual las dos partes comunicadas establecen y se aseguran mutuamente de sus identidades.

Estos mecanismos usan protocolos conocidos como **challenge-response**, como el de la figura 85:

1. El cliente A se presenta ante B indicándole su id A .
2. B le responde con un número aleatorio R_B , denominado *challenge*, sin encriptar. Esto es porque B aún no está seguro de si el mensaje recibido viene de A.
3. A le envía la aplicación de la clave compartida K_{AB} sobre el challenge R_B . Cuando B recibe esto, inmediatamente sabe que el mensaje proviene de A, dado que es el único que posee la clave K_{AB} .
4. Además, A le da a B su propio challenge, R_A , generado aleatoriamente.
5. Por último, B le responde a A aplicando la clave compartida K_{AB} al challenge R_A . Cuando A recibe esto, ahora él también está seguro de que está hablando con B.

A partir de ese momento, si ambos desean establecer una **clave de sesión** K_S , entonces A selecciona una y se la envía a B encriptada con K_{AB} .

Una forma de simplificar la secuencia de mensajes mencionada es hacer que cada participante transmita su identidad y el challenge en el mismo mensaje, sin esperar al envío correspondiente de la otra parte.

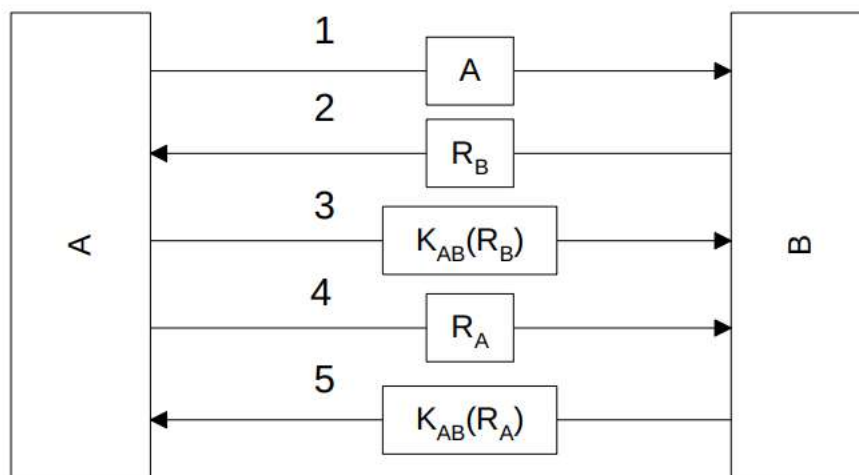


Figura 85: Protocolo challenge-response con clave secreta compartida.

10.7.2. Ataque por Sesiones

En los casos en los que se pueda establecer **sesiones múltiples** entre los participantes, se podría dar la siguiente secuencia:

1. En el mensaje 1, un tercero T simula ser A, enviando la identidad de A y el challenge R_T (versión simplificada mencionada antes).

2. En el mensaje 2, B responde como siempre con su challenge R_B , y espera que A se lo devuelva encriptado con K_{AB} .
3. En el mensaje 3, como T no conoce la clave K_{AB} , no puede devolver lo que espera B, por lo que inicia una segunda sesión y envía R_B como su challenge.
4. Cuando B le devuelve el challenge encriptado $K_{AB}(R_B)$ a T, en el mensaje 4, T utiliza esto como respuesta al mensaje 2 de la primera sesión, por lo que logra engañar a B. Acto seguido, aborta la segunda sesión, habiendo tenido éxito haciéndose pasar por A.

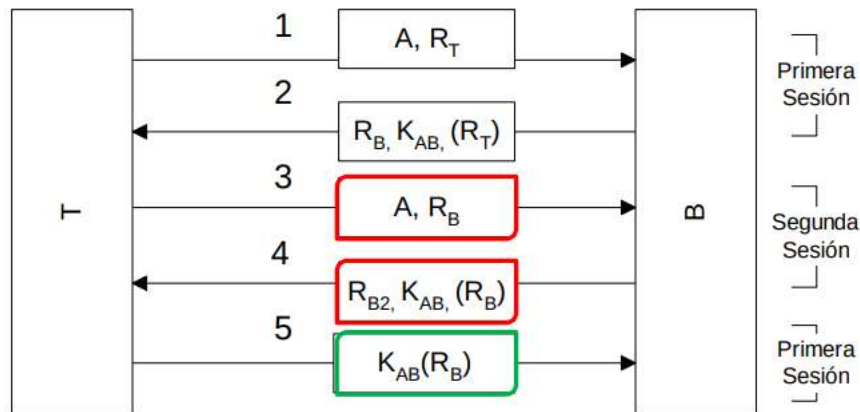


Figura 86: Ataque por sesiones en un protocolo challenge-response. T se interpone en la comunicación entre A y B, haciéndose pasar por A.

Reglas de diseño para evitar este tipo de problemas:

- Una posible manera de solucionar este problema es que B no acepte como challenge de un cliente algo que él ya haya enviado como challenge.
- El participante que inicia la transmisión debe probar su identidad antes que el receptor.
- Ambos participantes deben usar claves diferentes para la verificación de identidades.
- Deben elegir los challenges de conjuntos diferentes.

10.7.3. Distribución de Claves Simétricas Confiable

Los Key Distribution Centers (*centros de distribución de claves*, o KDC) son organismos intermedios confiables, que utilizan protocolos de acreditación y auditabilidad para asegurar que sus mecanismos son robustos.

Los mecanismos en cuestión implican que el emisor de una comunicación, A, selecciona una clave de sesión K_S , y le comunica al KDC su intención de comunicarse con el receptor B.

Este mensaje es encriptado utilizando la clave secreta K_A , que sólo conocen A y el KDC. Este último desencripta el mensaje tomando la identidad de B y la clave de sesión, y luego construye un nuevo mensaje con la identidad de A y la clave de sesión, que le envía a B encriptado con K_B , la clave secreta que sólo conocen B y el KDC.

Cuando B desencripta el mensaje con su clave, sabe que A se quiere comunicar con él, y sabe también la clave que éste desea utilizar.

10.7.4. Certificados Digitales

Los **certificados digitales** son relaciones válidas entre claves públicas y cierta entidad (por ejemplo una empresa).

Para darle validez a estos certificados, se confía en organizaciones como **autoridades certificadoras** (CA), o cadenas de confianza.

Por ejemplo, cuando Google quiere distribuir su clave pública, le solicita a una CA que firme digitalmente un certificado de clave pública suyo. Luego simplemente distribuye ese certificado firmado. Un usuario que confía en la CA, podrá corroborar que la clave pública de Google es la que firma el certificado.

10.8. Secure Sockets Layer (SSL) y Transport Layer Security (TLS)

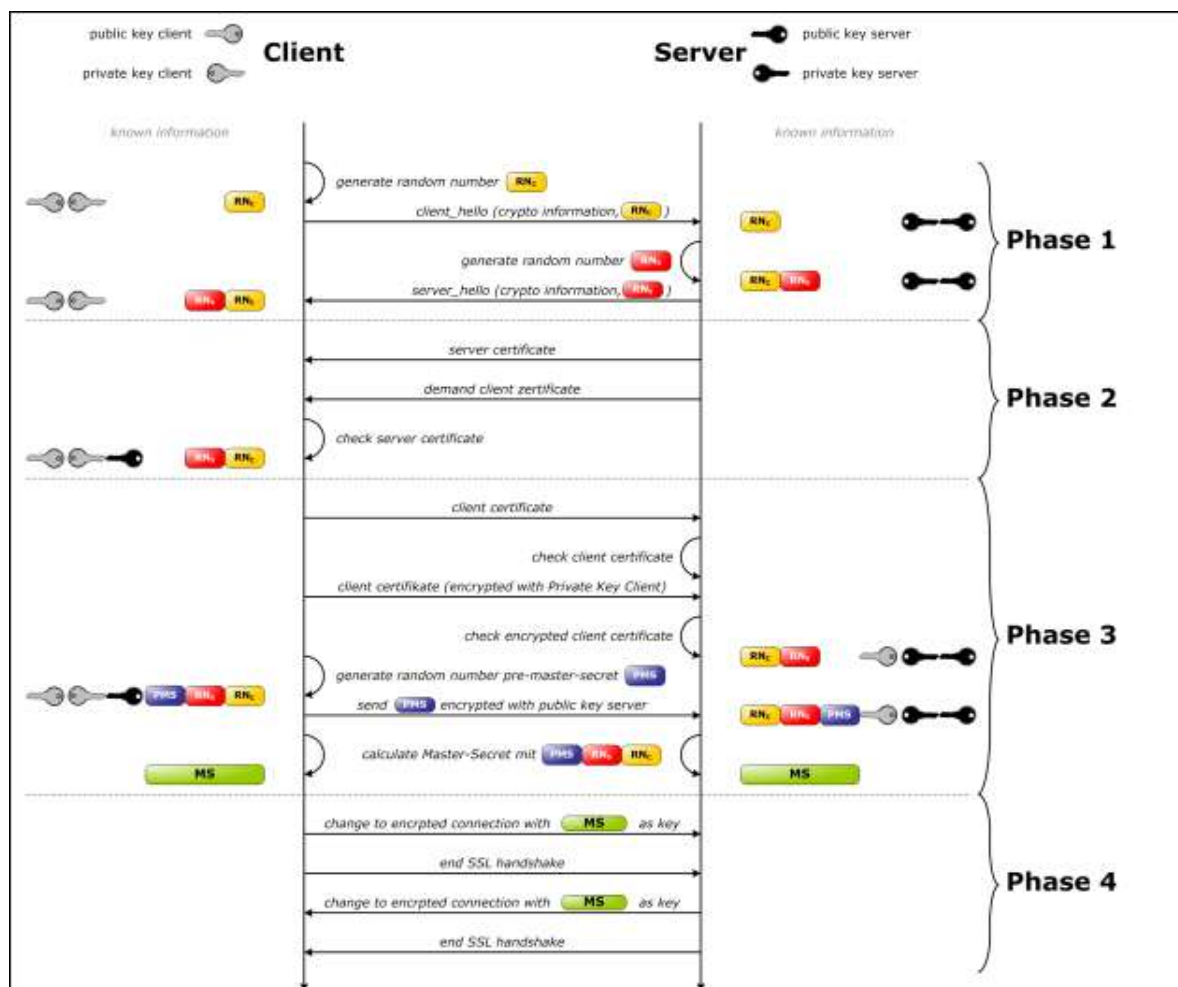


Figura 87: Handshake SSL.

Transport Layer Security (*seguridad de la capa de transporte*, o TLS) es el sucesor de Secure Sockets Layer (*capa de puertos seguros*, o SSL). Es un protocolo criptográfico diseñado para proveer seguridad a las comunicaciones sobre redes.

Principalmente apunta a proporcionar **privacidad** e **integridad** (mediante MAC) de datos entre las aplicaciones que se comunican. Corre en el **nivel de aplicación** y se compone de dos capas: el **record TLS** y los protocolos de **handshake TLS**.

Además, provee **autenticación** del servidor frente al cliente y, opcionalmente, del cliente frente al servidor. Esto se consigue con **certificados** de clave pública.

También de manera opcional, puede proporcionar **confidencialidad** a través del **cifrado** con un algoritmo simétrico.

10.8.1. Handshake SSL

1. Se negocian los algoritmos a utilizar durante la conexión.
2. Se autentica al servidor o a ambas partes.
3. Se genera un canal seguro para definir una clave maestra (master key).
4. Se derivan las claves necesarias a partir de la clave maestra.
5. Se constata la integridad de los mensajes de intercambio de claves.

10.9. Firewall

Un **firewall** es un conjunto de sistemas encargado de mediar todas las comunicaciones entre dos redes, implementando una política de seguridad de **control de acceso**, mediante el **filtrado de paquetes**.

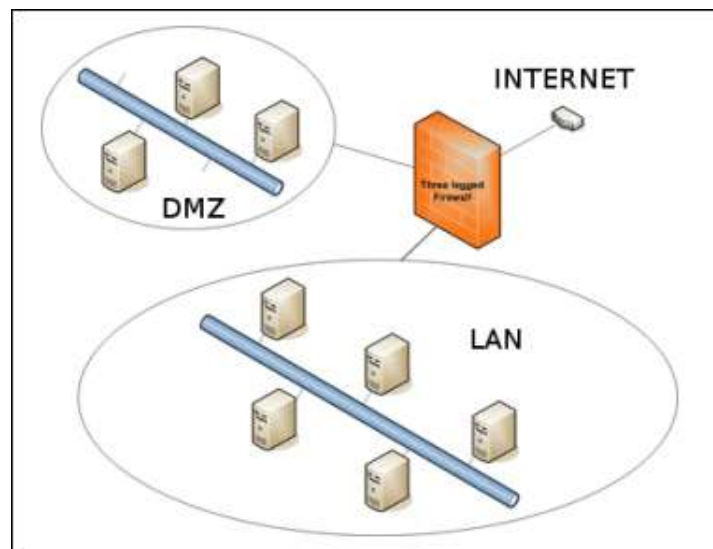


Figura 88: Three-legged firewall (*firewall de tres patas*): Internet, DMZ y LAN.

Tipos de firewall:

- **Stateful:** mantiene el estado de las conexiones entrantes y salientes, y puede permitir o denegar acceso teniendo en cuenta la sesión a la que pertenece.
- **Stateless:** utiliza reglas sencillas sobre los mensaje entrantes y salientes, filtrando paquetes según direcciones de origen y destino, puertos de origen y destino, y el protocolo utilizado.
- **Gateways de aplicación:** es un proxy inteligente; entiende ciertos protocolos, y sirve para aplicar filtro de capa de aplicación.

10.10. Ataques de Red

10.10.1. Sniffing

- Idea: **escuchar** los datos de la red, sin interferir en la conexión.
- Propósito: descubrir contraseñas y otra información confidencial.
- Protección: **encriptación de datos**.

10.10.2. Spoofing

- Idea: **hacerse pasar por otro**, interviniendo una conexión.
- Propósito: acceder a recursos confiados sin privilegios.
- Protección: **encriptación de protocolo**.

10.10.3. Hijacking

- Idea: **robar la conexión** después de autenticarse.
- Propósito: acceder a recursos no confiados sin privilegios.
- Protección: **encriptación de protocolo**.

10.10.4. Ingeniería Social

- Idea: **aprovechar** la buena voluntad de los usuarios.
- Propósito: tomar privilegios de otros usuarios.
- Protección: **autenticación** fuerte e información.

10.10.5. Explotar Bugs

- Idea: **aprovechar errores** de implementación en el software.
- Propósito: acceder a recursos sin privilegios.
- Protección: baterías de **tests** y listas CERT.

10.10.6. Confianza Transitiva

- Idea: **aprovechar** la confianza UNIX entre usuarios o hosts.
- Propósito: tomar privilegios de otros usuarios o hosts.
- Protección: **autenticación** fuerte y **filtrado** de paquetes.

10.10.7. Ataques Dirigidos por Datos

- Idea: ataque diferido **originado por datos recibidos**.
- Propósito: acceder a recursos sin privilegios.
- Protección: **firma digital** e información.

10.10.8. Caballo de Troya

- Idea: ataque diferido **originado por un programa recibido**.
- Propósito: acceder a recursos sin privilegios.
- Protección: **firma digital** e información.

10.10.9. Denegación de Servicio (DoS)

- Idea: **bloquear** un determinado conjunto de servicios.
- Propósito: que los usuarios legítimos no puedan usar los servicios.
- Protección: **hardware y software** especializado, firewalls.

10.10.10. Enrutamiento Fuente

- Idea: **modificar la ruta** de vuelta de los paquetes.
- Propósito: acceder a recursos confiados sin privilegios.
- Protección: **filtrado** de paquetes.

10.10.11. Adivinación de Contraseñas

- Idea: **probar** sistemáticamente contraseñas.
- Propósito: que los usuarios no legítimos puedan usar la identidad del usuario.
- Protección: **autenticación** fuerte.

10.10.12. Mensajes de Control de Red

- Idea: usar **mensajes ICMP** para aprovechar malas implementaciones de la pila TCP/IP.
- Propósito: acceder a paquetes de otra red.
- Protección: **filtrado** de paquetes.

10.11. Fuentes

- Rodrigo Castro. Teoría de las Comunicaciones, Clase Teórica 9. Primer cuatrimestre, 2020.
- Martín Medina. Teoría de las Comunicaciones, Clase Práctica 11. Primer cuatrimestre, 2020.
- Guido Tagliavini Ponce. Resumen de Teoría de las Comunicaciones. 2017.